Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Opening up architectures of software-intensive systems

A functional decomposition to support system comprehension

P. Charland
D. Ouellet
D. Dessureault
M. Lizotte
DRDC Valcartier

Defence R&D Canada – Valcartier

Technical Memorandum

DRDC Valcartier TM 2006-732

October 2007

Canada

Opening up architectures of software-intensive systems

A functional decomposition to support system comprehension

P. Charland
D. Ouellet
D. Dessureault
M. Lizotte
DRDC Valcartier

Defence R&D Canada - Valcartier

Technical Memorandum

DRDC Valcartier TM 2006-732

October 2007

Author

Philippe Charland

Approved by

Guy Turcotte
Head System of Systems

Approved for Release by

Philip Twardawa
Chief Scientist

Abstract

With the increasing needs of the Canadian Forces (CF) for systems interoperability, techniques and tools have to be developed in order to build systems of systems (SoS), i.e., systems whose components are themselves independent systems from an operational and managerial viewpoint. However, before existing systems can interoperate, their architectures first need to be recovered and comprehended. This technical memorandum describes the functional decomposition of an integrated suite of tools to assist with software system architecture recovery and comprehension. It was designed based on the requirements already identified in the scientific literature for comprehension tools, on a qualitative study conducted using existing tools, as well as on a state-of-the-art survey on system architecture recovery and comprehension. Following the conception of this functional decomposition, a prototype implementing it will be developed into an integrated development environment (IDE) to assist the CF in recovering and comprehending the architecture of already existing software systems.

Résumé

Avec les besoins croissants des Forces canadiennes (FC) en matière d’interopérabilité de systèmes, des techniques et outils ont besoin d’être développés afin de construire des systèmes de systèmes (SdS), c’est-à-dire des systèmes dont les composantes sont elles-mêmes des systèmes indépendants d’un point de vue opérationnel et de gestion. Cependant, avant que des systèmes existants puissent interopérer, leurs architectures ont d’abord besoin d’être récupérées et comprises. Le présent mémorandum technique décrit la décomposition fonctionnelle d’une suite d’outils intégrés pour aider à la récupération et la compréhension d’architectures de systèmes logiciels. Elle a été conçue en se basant sur les exigences déjà identifiées dans la littérature scientifique pour les outils de compréhension, sur une étude qualitative menée en utilisant les outils existants, ainsi que sur une revue de l’état des connaissances portant sur la récupération et la compréhension d’architectures de systèmes. À la suite de la conception de cette décomposition fonctionnelle, un prototype l’implantant sera développé dans un environnement de développement intégré (EDI) afin d’aider les FC à récupérer et comprendre les architectures de systèmes logiciels déjà existants.

This page intentionally left blank.

Executive Summary

Over the years, the needs of the Canadian Forces (CF) for systems interoperability have significantly increased. As the CF demand greater systems interoperability, their software architects need techniques and tools to comprehend the architecture of existing systems before making them interoperate in order to build systems of systems (SoS). Although some requirements have already been identified in the scientific literature for comprehension tools, these are not specifically targeted for the understanding of software systems at the architectural level. This technical memorandum describes the functional decomposition of an integrated suite of tools to assist with the recovery and comprehension of software systems architectures. It was designed by the members of the Opening up Architectures of Software-Intensive Systems (OASIS) project. It is based, to some extent, on the requirements one can find in the open literature. It is also based on the results of a qualitative study conducted to assess the value added by existing analysis tools on the understanding of participants performing comprehension tasks, at the architectural level, on large scale military applications. Finally, it takes into consideration the findings contained in a state-of-the-art survey on system architecture recovery and comprehension that was carried out as a previous phase of the OASIS project.

The functional architecture presented in this technical memorandum consists of the following ten subsystems:

Repositories Store persistently the facts extracted about a system under study.

Data Access Provides an interface between the meta-model used by the Information Management Services subsystem and the Repositories.

Information Management Services Provides a meta-model to facilitate the integration of information producers and consumers as well as the discovery of available information.

Fact Extraction Fundamental step of architecture recovery and comprehension that consists of finding pieces of information about a system under study.

Analysis Allows to separate a system into its constituent parts.

Synthesis Provides the capacity to combine several extracted facts to form a new whole at a higher level of abstraction.

Visualization Uses graphical techniques to make a software system visible through the display of its artifacts and behavior.

Documentation Generation Produces system documentation at different levels of abstraction from the extracted facts.

Comprehension Process Provides guidance to users in recovering and comprehending the architecture of a software system.

Graphical User Interface Allows users to interact with and control the architecture recovery and comprehension tool in a highly visual manner.

The above functional decomposition is a way to synthesize the knowledge of the OASIS research group on architecture recovery and comprehension. It serves as a reference model which is destined to evolve with the advancement of the group's knowledge in this research area and orient its future work.

Following the conception of the functional decomposition previously described, a prototype implementing a subset of it will be developed into an integrated development environment (IDE) to assist the CF in recovering and comprehending the architecture of already existing software systems. Ideally, once this prototype is developed, another study, similar to the qualitative study previously conducted, but with an improved design and set of comprehension tasks, should be performed. Its objective would be to assess the added value of the OASIS architecture recovery and comprehension prototype on the understanding of participants.

Charland, P., Ouellet D., Dessureault, D., Lizotte M. 2007. Opening up architectures of software-intensive systems: A functional decomposition to support system comprehension. DRDC Valcartier TM 2006-732. Defence R&D Canada - Valcartier.

Sommaire

Au cours des années, les besoins des Forces canadiennes (FC) en matière d'interopérabilité de systèmes ont augmenté de façon significative. Alors que les FC exigent plus d'interopérabilité entre les systèmes, leurs architectes logiciels ont besoin de techniques et d'outils pour comprendre l'architecture des systèmes existants avant de les faire interopérer pour construire un système de systèmes (SdS). Bien que des exigences aient déjà été identifiées dans la littérature scientifique pour des outils de compréhension, celles-ci ne sont pas expressément ciblées pour la compréhension de systèmes logiciels sur le plan de l'architecture. Le présent mémorandum technique décrit la décomposition fonctionnelle d'une suite d'outils intégrés pour aider à récupérer et comprendre les architectures de systèmes logiciels. Celle-ci a été conçue par les membres du projet Ouverture d'Architectures de Systèmes Informatisés Significativement (OASIS). Elle est basée, jusqu'à un certain point, sur les exigences qui peuvent être trouvées dans la littérature ouverte. Elle est aussi fondée sur les résultats d'une étude qualitative qui a été menée afin d'évaluer la valeur ajoutée d'outils d'analyse sur la compréhension de participants accomplissant des tâches de compréhension sur des applications militaires de grande taille, sur le plan de l'architecture. Finalement, cette décomposition fonctionnelle tient compte des conclusions contenues dans une étude de pointe sur la récupération et la compréhension d'architectures de systèmes qui a été menée lors d'une phase précédente du projet OASIS.

L'architecture fonctionnelle présentée dans ce mémorandum technique est composée des dix sous-systèmes suivants :

Référentiels Enregistrent de façon persistante les faits qui ont été extraits à partir du système à l'étude.

Accès aux données Fournit une interface entre le méta-modèle utilisé par le sous-système Services de gestion de l'information et les Référentiels.

Services de gestion de l'information Fournit un méta-modèle afin de faciliter l'intégration de producteurs et de consommateurs d'information ainsi que la découverte d'information disponible.

Extraction de faits Étape fondamentale de la récupération et la compréhension d'architectures qui consiste à trouver des fragments d'information au sujet d'un système à l'étude.

Analyse Permet de séparer un système en ses parties constituantes.

Synthèse Fournit la capacité de combiner plusieurs faits qui ont été extraits pour former un nouveau tout à un niveau d'abstraction supérieur.

Visualisation Utilise des techniques graphiques afin de rendre visible un système logiciel par l'entremise de l'affichage de ses artefacts et de son fonctionnement.

Génération de documentation Produit la documentation du système à différents niveaux d'abstraction à partir des faits qui ont été extraits.

Processus de compréhension Guide les usagers pour récupérer et comprendre l'architecture d'un système logiciel.

Interface usager graphique Permet aux usagers de contrôler et d'interagir avec l'outil de récupération et de compréhension d'architectures de manière très visuelle.

La décomposition fonctionnelle ci-dessus est une façon de synthétiser la connaissance du groupe de recherche OASIS en récupération et compréhension d'architectures. Elle sert de modèle de référence qui sera appelé à évoluer avec l'avancement des connaissances du groupe dans ce domaine de recherche ainsi qu'à orienter ses travaux futurs.

Suite à la conception de la décomposition fonctionnelle décrite précédemment, un prototype implantant un sous-ensemble de celle-ci sera développé dans un environnement de développement intégré (EDI) afin d'aider les FC à récupérer et comprendre les architectures de systèmes logiciels déjà existants. Idéalement, une fois que ce prototype sera développé, une autre étude, similaire à l'étude qualitative précédente, mais avec une conception et une série de tâches de compréhension améliorées, devrait être menée. Son objectif serait d'évaluer la valeur ajoutée du prototype de récupération et de compréhension d'architectures OASIS sur la compréhension des participants.

Charland, P., Ouellet D., Dessureault, D., Lizotte M. 2007. Opening up architectures of software intensive-systems: A functional decomposition to support system comprehension. DRDC Valcartier TM 2006-732. R&D pour la défense Canada - Valcartier.

Table of Contents

List of Figures

List of Tables

1. Introduction

Over the years, the needs of the Canadian Forces (CF) for systems interoperability have significantly increased. For example, to improve the automation of the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) process, a large number of software intensive systems must interact together to handle a massive amount of information. The CF also require systems interoperability when they collaborate with allied nations to achieve common objectives.

As the CF demand greater systems interoperability, their software architects need techniques and tools to understand the architecture of existing systems and make them interoperate in order to build a system of systems (SoS). A SoS is an assemblage of components which individually may be regarded as systems and which possess two additional properties: operational and managerial independence of the components [1]. Each component system must be able to operate independently if the SoS is disassembled. Furthermore, even though the component systems are separately acquired and integrated, they maintain a continuing operating existence independent of the SoS. An example of a SoS is a system built for a coalition operation, where each participating nation brings its own operational planning system.

Before existing systems can interoperate, their architectures first need to be understood. The architecture of a system can be defined as the structure of its components, their interrelationships, as well as the principles and guidelines governing their design and evolution over time [2]. However, understanding the architecture of systems can prove to be quite a complex task. These systems have most probably undergone several code revisions without a real concern about maintaining their architectural design documentation up to date [3]. As a result, architecture recovery has to be performed to regenerate coherent abstractions and guide architects during their comprehension task. Architecture recovery can be described as the process of retrieving up-to-date architectural information from existing source code artefacts. The rational of system architectural recovery is to provide reasoning behind the software architecture or high-level organization of a system.

To support the effort of developing methodologies, techniques, and tools needed for the recovery and comprehension of existing systems architecture, the SoS section of Defence Research and Development Canada (DRDC) Valcartier started a project called Opening up Architectures of Software-Intensive Systems (OASIS) [4]. Its objective is to develop technical solutions in order to reduce the time needed to comprehend systems to be integrated into a SoS.

In a previous phase of the OASIS project, a state-of-the-art survey [5] of the current techniques and tools for architecture recovery and comprehension was carried out. Following this survey, a qualitative study was conducted to assess the added value of a selected subset of the tools previously identified on the understanding of participants performing high-level comprehension tasks on large-scale military systems [6].

The present technical memorandum describes the functional decomposition of an integrated suite of tools for architecture recovery and comprehension. It will address the limitations of the existing tools which were identified in the state-of-the-art survey and as part of the qualitative study. The remainder of this technical memorandum is organized as follows: Section 2 presents an overview of the comprehension models. In Section 3, the factors affecting comprehension are provided. In Section 4 and 5, the implications of cognitive models on tool requirements are discussed. Section 6 presents the limitations of existing tools for architecture recovery and comprehension. In Section 7, the different subsystems of the OASIS functional architecture are described. Finally, Section 8 provides conclusions and future work.

2. Cognitive Models of Program Comprehension

Several studies have been conducted to determine which strategies programmers use when trying to understand unfamiliar code. The results have demonstrated that different cognitive models are applied to create mental representations of programs under examination. But before these models can be reviewed, their terminology first needs to be defined.

2.1 Concepts and Terminology

A programmer's mental representation of a program under study is referred as the *mental model* [7]. The cognitive processes and temporary information structures used by the programmer to form the mental model are described by a *cognitive model* [7].

Programming plans are generic fragments of source code which represent typical programming scenarios. An example of a programming plan is a sorting algorithm [8]. *Delocalized plans* are pieces of source code which are conceptually related, but physically located in non-contiguous parts of a program [9]. An example of a delocalized plan is the "retrieve-a-record & process-the-record" plan in a database management system. The first part of the plan is located in the `SEARCH` routine, while the second part is located, non-contiguously, in the `DELETE` routine [9].

Beacons are familiar features in the source code which act as cues to the presence of certain structures [10]. An example of a beacon is the swapping of two variables in a sorting algorithm. *Rules of programming discourse* are the programming conventions and algorithm implementations [8].

2.2 Cognitive Models

Following is an overview of some of the influential cognitive models in program comprehension as reviewed by Storey et al. [11].

2.2.1 Bottom-Up

Shneiderman [12,13] proposed that programs are understood bottom-up, i.e., by first reading the source code and then mentally grouping lower level software artifacts into higher level abstractions that are more meaningful. These abstractions are further aggregated until a high level comprehension of the program is obtained. The cognitive framework of Shneiderman and Mayer [12] makes a distinction between the syntactic and semantic knowledge of a program. The syntactic knowledge is language dependent and relates to the statements of a program, while the semantic knowledge is language independent and is formed in progressive layers until a mental model of the application domain is built.

In [14], Pennington also observed that programmers use a bottom-up strategy when trying to understand a program. They first produce a control flow abstraction, referred as the program model, which represents the sequence of operations of the program. This model is generated by grouping source code microstructures (statements, predicate statements, dependencies) into macrostructures (source code structure abstractions) and then by cross-referencing them. After the program model has been assimilated, the situation model is generated. This model incorporates knowledge about the data flow and the functional abstractions, e.g., the program goals hierarchy.

2.2.2 Top-Down

Brooks formed a theory that programs are understood in a top-down manner, where the knowledge about the application domain is first reconstructed and then mapped on the source code [10]. This process starts with the formulation of a hypothesis about the general nature of the program. This global hypothesis is then refined into a hierarchy of secondary hypotheses, which are evaluated in a depth-first manner. The validation of rejection of a hypothesis depends heavily on the presence or absence of beacons [10].

Soloway and Ehrlich [8] observed that a top-down strategy is used when the source code or type of source code is familiar. They also noted that experienced programmers use beacons, programming plans, as well as rules of programming discourse in order to decompose goals and plans to a lower level. Furthermore, it was observed that delocalized plans complicate program comprehension, as they involve finding causal interactions between non-contiguously located pieces of source code.

2.2.3 Knowledge-Based

Letovsky [15] suggested that programmers are opportunistic processors, capable of understanding programs using either a bottom-up or top-down approach, depending on the cues available. His theory has three components: a knowledge base, which encodes the programmer's expertise and knowledge about the application; a mental model, which represents the programmer's current understanding of the program; and an assimilation process, which explains how the mental model evolves using the knowledge base and information about the program.

Inquiry episodes are an essential part of the assimilation process. During such an episode, a programmer asks a question, forms a hypothesis, and searches through the source code and documentation to validate or reject the hypothesis. Inquiry episodes often happen as a result of delocalized plans.

2.2.4 Systematic and As-Needed

In [16], Littman et al. observed programmers enhancing a personnel database program. They noted that the programmers either read the source code systematically, tracing the control and data flow dependencies in order to acquire a general understanding, or used an as-needed approach, focusing only on the source code related to the task to achieve. The subjects using a systematic approach gained information about the structure of the program and the interactions between its components at run-time. The ones who used an as-needed approach only acquired static knowledge, resulting in a weaker mental model compared to the one of the other subjects. They also made more errors, as they did not identify the dynamic interactions between the components.

Soloway et al. [9] combined these two theories as macro-strategies in order to understand programs at a more global level. Using this strategy, the programmer traces the flow dependencies for the whole program and performs simulations as the source code and documentation are read. However, this method is not applicable for programs of considerable size. In the more commonly used approach, programmers examine only what they consider relevant. The drawback of this approach is that more mistakes can be made, since important interactions can be missed.

2.2.5 Integrated Metamodel

Based on the results of experiments, Von Mayrhauser and Vans combined the previous approaches into a single metamodel [17]. They suggested that understanding is built at several levels of abstractions, by freely switching between the different comprehension strategies. Their model is composed of four components. The first three detail the comprehension processes used to create the mental representations at different levels of abstractions. The fourth component describes the knowledge base used to carry out the comprehension process. In their integrated metamodel [17]:

- The top-down approach is invoked as an as-needed strategy, when the source code or programming language is familiar. It uses the domain knowledge as a starting point for the formulation of hypotheses.

- The program model, which is a control flow abstraction, is invoked when the source code and application are completely unfamiliar.

- The situation model, which describes the data flow and functional abstractions in a program, is developed after a partial program model has been formed using systematic or opportunistic strategies.

- The knowledge base contains the information required to build these three cognitive models. It stores the programmer’s current knowledge as well as the one acquired and inferred during the comprehension process.

3. Factors Affecting Comprehension

The wide variety of cognitive models discussed previously stems from the fact that certain factors will affect how a programmer tackles a comprehension task [18, 19]. These factors are the program under study, the characteristics of the programmer, as well as the comprehension task to achieve. These factors are discussed next, as summarized by Storey [7].

3.1 Program Characteristics

As one would normally expect, programs which are well designed and documented are easier to understand than badly designed and documented ones. However, the programming language in which an application is written also affects comprehension, as was shown is Pennington's experiment [14].

Object-oriented languages are often perceived to offer a more natural correspondence with real world problems, due to the inheritance and association relationships [20]. However, others argue that object-oriented programs are difficult to understand, as they involve a very strong delocalization of plans: a plan may be distributed through several procedures, each attached to a different class [20].

3.2 Individual Programmer Differences

With experience, programmers recognize which strategy is the most efficient for a given program and comprehension task [11]. Détienne also observed that experienced programmers make more use of external devices as memory aids [20]. Furthermore, in [21], Vessey noted that they tend to reason about programs according to both functional and object-oriented relationships, as well as consider algorithms, while programmers with less experience tend to focus on objects. These observations highlight the fact that program comprehension tools should enhance or ease the programmer's preferred strategies, rather than impose a fixed one, which may not always be suitable [11].

Although experience influences the comprehension strategy adopted by a programmer, it is not the only factor to consider when elicitating requirements for a supporting tool. For example, programmers' ability and creativity, which cannot simply be measured by their experience, also affect how they will address a comprehension task [7].

3.3 Task Variability

Program comprehension is not an end goal by itself. On the contrary, it is the necessary first step towards the realization of other objectives, such as correcting a fault, reusing source code, or adding functionalities to a software system. The type and scope of the end objective will influence the comprehension process followed by a programmer. For example, a simple task might only require understanding a small portion of source code, while a more complex one might necessitate taking into account global interactions. As a result, the programmer will have to acquire an in-depth comprehension of the causal relationships for the whole program [7].

In [14], Pennington explained that when a programmer has to perform a task which necessitates recall and comprehension, he or she will form a program model, i.e., the sequence of operations of the program. In the case the task requires making a modification to a program, the programmer will form a situation model, which incorporates knowledge about the data flow and the functional abstractions.

A task which involves the reuse of source code requires that a programmer first understands the problem to solve. Then, a suitable solution has to be retrieved from the existing source code and adapted to the current problem. The mapping of the problem to the solution is often done using analogical reasoning [20] and might involve iterative searching through many possible solutions [7].

4. Cognitive Models and Tool Implications

In this section, the requirements that comprehension tools should implement are presented in general terms. It should be noted that they should be fine-tuned for the specific settings in which the tool implementing them is deployed. These requirements, as reviewed by Storey [7], are based on the cognitive models introduced in Section 2.

4.1 Browsing Support

Browsing allows programmers to navigate through the source code, according to the dependencies between the different software elements, such as the definitions of variables, methods, classes and their use. Since definitions and uses may be distributed over distant parts in the source code, browsing accelerates the navigation and lessens the effort needed for software comprehension [22].

Both the top-down and bottom-up comprehension models necessitate browsing support. On the one hand, the top-down cognitive model requires browsing the source code of an application, by starting with the high-level abstractions or concepts and going down to the lower level details, using the presence of beacons. On the other hand, bottom-up comprehension necessitates tracking the control and data flow chains in order to form both the program and situation models.

For the integrated metamodel, both top-down and bottom-up browsing should be supported, as the strategy used depends on the available cues. Also, having both breadth-first and depth-first browsing support available should alleviate the difficulties posed by delocalized plans introduced by object-oriented languages.

4.2 Searching

While browsing is an exploratory strategy, without a fixed endpoint and relatively unstructured, searching is a planned activity with a specific goal. An example of searching is to look for pieces of source code, as in the case of a code reuse task, or when enquiring about the role of a variable, method, or class in a program.

4.3 Multiple Views

A comprehension tool should provide different views of a program under study. It should offer static views, which show the structure of a system, in particular, the kinds of things that exist, e.g., classes, their internal structures, and their relationships to other things [23]. To complement the static views, dynamic views should also be provided. These views show the behavior of a system, such as the objects and threads that exist at run-time, as well as the method invocations between them.

4.4 Context-Driven Views

In addition to provide multiple views, the tool should also offer the possibility to select the view which is the most appropriate to the current context. For example, in the case of an object-oriented program, it is generally more suitable to display the inheritance hierarchy as the initial view. However, in the case the structure of the latter is rather flat, the user may prefer to first view the call graph [7].

5. Tool Requirements Explicitly Identified

The previous section enumerated in general terms the functional requirements a program comprehension tool should have. In this section, specific requirements for comprehension tools are listed. This list comes from [7], in which the efforts conducted by several researchers are surveyed.

5.1 Biggerstaff

In [24], Biggerstaff et al. mention that the main challenge in software understanding is the concept assignment problem, i.e., the mapping of source code elements to their corresponding requirements. Even though automated techniques can assist in the location of programming concepts and features in the source code, the difficulty comes from automatically locating human oriented concepts. According to Biggerstaff, the most important functional requirements for comprehension tools are queries, graphical views, and hypertext.

5.2 Von Mayrhausser and Vans

Following their research on the integrated metamodel, Von Mayrhausser and Vans determined basic information needs for reverse engineering tasks [17]. They also suggested some functional requirements to satisfy them, according to the different models [17]:

- Top-Down Model: On-line documents with keyword searching capacities; call graph pruning based on specific categories; smart differencing features; browsing of locations history; entity fan-in.
- Situation Model: Complete list of domain sources, including non source-code based; visual representations of most important domain functions.
- Program Model: Pop-up declarations; online cross-referencing reports; function counting.

5.3 Singer and Lethbridge

In [25], Singer et al. observed the work practices of software engineers in a company. Their study was conducted at the individual, group, and company-wide levels. Following their observations, they recommended that a tool should support “just-in-time comprehension of source code.” This recommendation was based on the fact that engineers tend to forget quickly the specificities of a program part when moving to another one. This constrained them to comprehend again this part when it was needed by the task they had to accomplish. To support just-in-time comprehension, a tool should offer the following functional requirements [25]:

- The capacity to search through the source code by either specifying names or patterns.

- The capacity to display the pertinent attributes of the elements retrieved by a search as well as the relationships between them.

- Features to store persistently the searches and problem-solving sessions and support their navigation.

These requirements were implemented in the tkSee tool, which became used by the company's engineers.

5.4 Erdös and Sneed

Following many years spent in the industry doing maintenance and reengineering work, Erdös and Sneed implemented a tool to support the maintenance of applications for which programmers had an incomplete understanding. The tool provided answers to the following questions, proposed by Erdös and Sneed in such situations [26]:

1. Where is a partial subroutine/procedure invoked?

2. What are the arguments and results of a function?

3. How does control flow reach a particular location?

4. Where is a particular variable set, used or queried?

5. Where is a particular variable declared?

6. Where is a particular data object accessed?

7. What are the inputs and outputs of a module?

It should be noted that today's IDEs (e.g., Eclipse [27], Visual Studio 2005 [28]) offer functionalities which provide answers to most of the above questions.

6. Limitations of Existing Tools

As mentioned in the introduction, in a previous phase of the OASIS project, a state-of-the-art-survey on the current techniques and tools for architecture recovery and comprehension was carried out [5]. The limitations of the tools surveyed are discussed next.

6.1 Multi-Language Support

Source code based architecture recovery is derived from parsing which provides static structural information about the source code to be analyzed. There exist several techniques that can be applied on the parsed information to extract further dependencies in the source code to support the architecture recovery and comprehension process. However, most of the parsers of the currently existing tools support systems written in a single programming language [5].

6.2 Static Analysis Support

Analytical support is essential to comprehend source code and implementation dependencies. However, the vast majority of the architectural recovery tools surveyed lack detailed analytical support. Their analysis functionalities are mainly static and limited to standard high-level dependencies and metrics. According to the author of the survey, the integration of additional techniques, such as concept analysis, would be beneficial for architecture comprehension. Furthermore, the extraction, grouping, and clustering of the available information from knowledge bases would also benefit from a more detailed analysis of the dependencies among the different source code components [5].

6.3 Dynamic Analysis Support

An increasing part of today's legacy software systems are object-oriented and/or distributed [5]. The use of constructs such as inheritance, polymorphism, and dynamic binding results in the fact that the exact behavior of a system is only known at run-time. Under these circumstances, static analysis alone is insufficient to recover and comprehend the architecture of a subject system. It has to be complemented by dynamic analysis. Unfortunately, most current architecture recovery tools focus on static analysis of source code, which therefore limits their applicability for the architectural recovery of object-oriented and distributed legacy systems [5].

6.4 Dynamic Visualization and Abstraction

The graphical descriptions of software architectures generated by current tools often focus on static calls and data relationships gathered by parsing the source code. These types of architecture graphs can exhibit extremely high connectivity and possess little contextual information with respect to the nature of the relationships between components [5]. Dynamic visualization is useful to understand higher-level system behavioral characteristics that cannot be determined from static architectural views. Some existing tools support dynamic visualization and structure querying, but at the object level. Therefore, the visualizations they provide are hard to scale and interpret for large and distributed applications [5].

6.5 Domain Knowledge

The last limitation that was identified in the state-of-the-art-survey is that the current tools do not offer functionalities to incorporate domain and user knowledge about a software system. Such knowledge is required to reconstruct and understand an application at the architectural level, as it allows mapping source code elements to their corresponding operational concepts.

7. OASIS Functional Architecture

As mentioned in Section 4, the requirements a comprehension tool should implement depend on the specific objectives to be achieved by its users. In the present case, the objective of the OASIS project is to reduce the time needed to comprehend systems to be integrated into a SoS. Therefore, the technical solutions developed as part of OASIS should assist in performing comprehension tasks at the architectural level with a focus on systems interoperability.

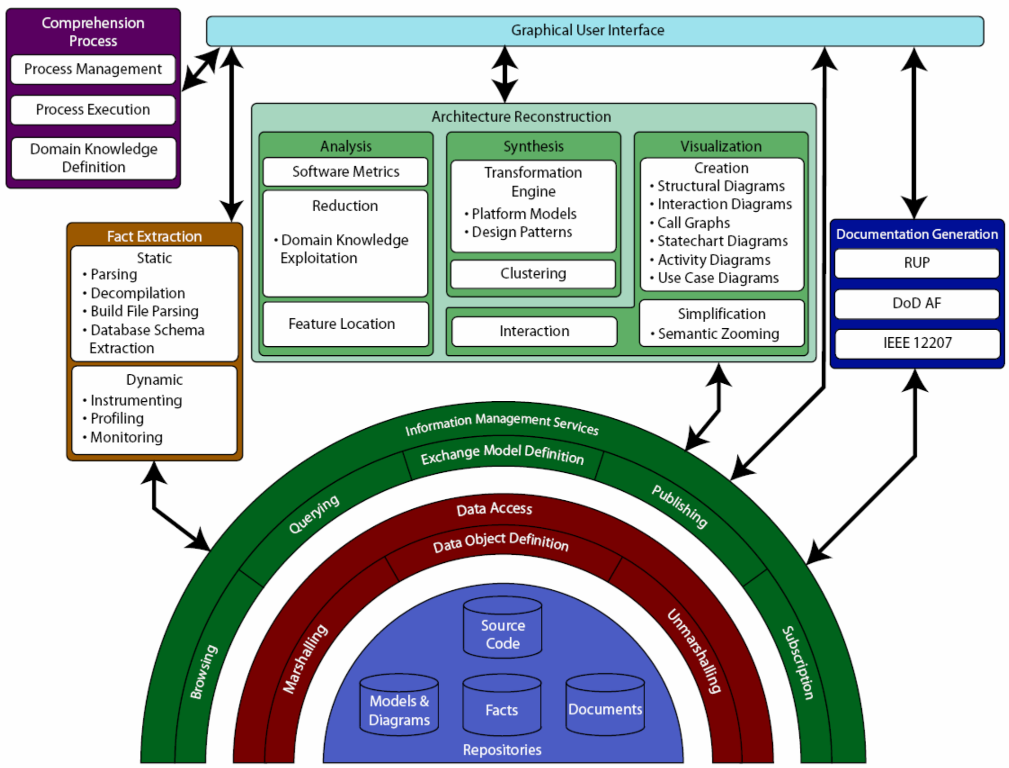


Figure 1. *OASIS Functional Architecture*

The above figure shows a visual representation of the OASIS functional architecture. As indicated, it consists of the following subsystems: Repositories, Data Access, Information Management Services, Fact Extraction, Analysis, Synthesis, Visualization, Documentation Generation, Comprehension Process, and Graphical User Interface. The functionality of each of these subsystems and their related service groups are further described in the remaining sections of the present document.

The OASIS functional decomposition was designed with the goal of recovering and comprehending the architectures of military applications written in C++ or Java and consisting of more than 1,000 classes. It takes into consideration the requirements for comprehension tools identified in Section 4 and 5, as well as the results of the qualitative study conducted by Charland et al. [6]. It also addresses the limitations of existing tools outlined in Section 6.

7.1 Repositories

In order to understand an existing software system, one needs to have access to different types of information. In the course of the comprehension process, additional information will also be generated. This information needs to be stored persistently in repositories. The OASIS functional architecture contains four such repositories, which store respectively the source code, facts, models and diagrams, as well as the documents associated with the software system under study. These repositories are logical ones and will not necessarily be implemented as databases.

7.1.1 Source Code

Most high-level reverse engineering analysis and architecture recovery activities are based on the software system source code. It is therefore one of the types of information which has to be stored in the repositories. There exist different mechanisms to manage source code and its associated files. These are described next.

7.1.1.1 File System

Using the operating system's file system to manage source code is the most simplistic approach, both in terms of implementation and functionality provided. It limits the amount of metadata that can be associated with a source code file. The specific metadata elements are determined by the file system, but usually consist of fields such as the file owner and the date of the last modification. These elements can be used to assist with the comprehension of a software system, but this information if often lost when the source code is transferred to another computer, as the date of the last modification will be set to the date of the copying [29]. This approach is also limited by the fact that only the latest version of the source code is stored, as the history of changes made to source files is not captured.

7.1.1.2 Versioning Systems

Versioning systems such as CVS (Concurrent Versions System) [30] provide additional information over the file system approach. As implied by the name, they keep track of all changes in a set of files. A complete version of the source code is checked-out when needed by a programmer, and later checked-in when the work on the copy is completed. Information typically stored by versioning systems includes the author of the modification, the date, and comments about the revision.

The information provided by versioning systems can be useful when trying to understand an unfamiliar software system. For example, in [31], the CVS history of a repository is analyzed to extract information regarding the nature of the collaboration between team members. In [32], the change history is used to locate the merging and splitting of files and functions in procedural code. The objective is to recover information about the intent of a design change.

7.1.1.3 Integrated Development Environments

Integrated development environments (IDEs) such as Eclipse [27] can be seen as a complementary approach over the previous two, since they can be used with both. Although IDEs do not supply additional information to assist in understanding an unfamiliar software system, they provide functionalities to access its source code. These functionalities can in turn be used by a developer implementing a tool for architecture recovery and comprehension.

In Eclipse, the source code can be imported from a file system or CVS repository. Once imported, the Eclipse application programming interface (API) can be used to access the source code elements. For example, as part of the Java Development Tools (JDT) subproject, Eclipse has a Java model. In this model, compilation units, which implement the `ICompilationUnit` interface, represent Java source files. This is illustrated in Figure 2 below.

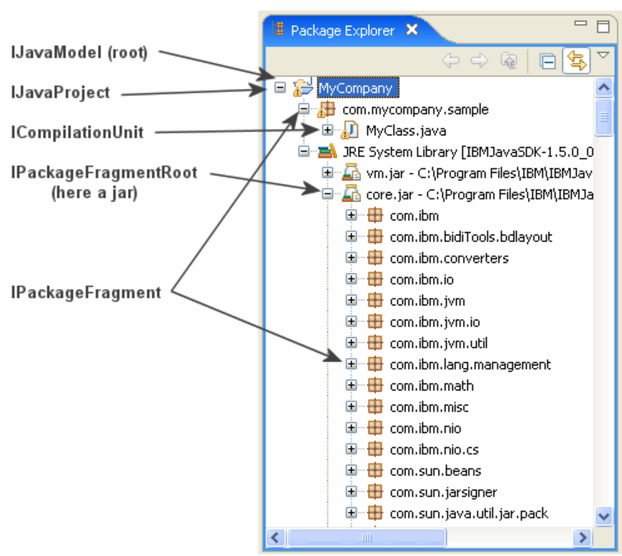


Figure 2. *Eclipse Java Model*

From the compilation units and using various interfaces (e.g., `IPackageDeclaration`, `IImportDeclaration`, `IType`, `IField`, and `IMethod`), the other elements declared in source files can be accessed, as shown in Figure 3. This allows developers implementing architecture recovery and comprehension tools to access the different elements of the source code.

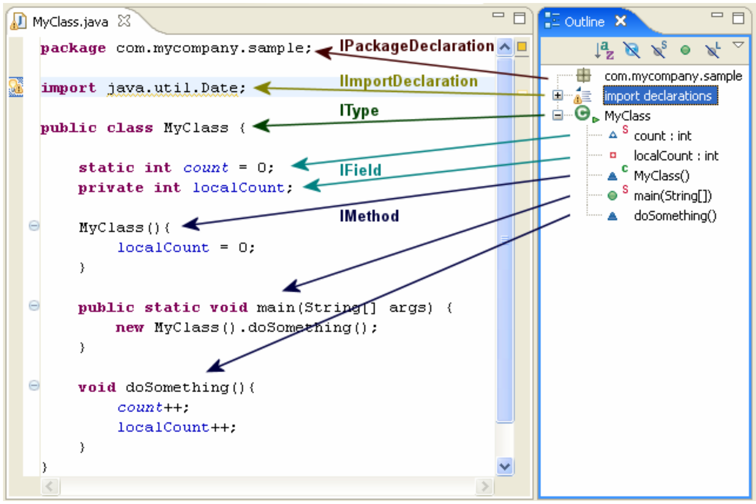


Figure 3. *Eclipse Java Model at the Source Code Level*

7.1.2 Facts

The Facts repository contains the basic facts about a subject system, at a low level of abstraction. These facts are usually extracted using, for example, lexical or parser-based tools, in the case of static information, or profiling tools for dynamic information. As each tool usually has its own specific data schema, the interoperability between tools is limited and often restricted to the use of a standard exchange format, such as the Graph eXchange Language (GXL), to describe the schema in the case of graph-based tools.

The Object Management Group (OMG) Architecture-Driven Modernization (ADM) Task Force [33] is looking at the above interoperability problem. It aims at producing a set of standards to describe information that can be extracted from existing systems to support modernisation efforts and facilitate interoperability between tools. However, there will always be existing tools to be integrated within OASIS that will not follow such a standard when it will become available. Therefore, the present functional architecture has a mechanism, described in more detail in Section 7.2 and 7.3, to allow existing tools to be integrated within OASIS using their own data schema.

7.1.3 Models and Diagrams

This repository stores persistently the diagrams, their models, as well as the associated metamodels, that are generated during the software architecture recovery and comprehension process. Examples of such diagrams are the ones contained in the UML specification. The relationships between a metamodel, model, and diagram are explained next and illustrated in Figure 4.

A metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest. For example, in UML, the metamodel defines the complete semantics for representing models using UML. In the case of a class diagram, the metamodel contains elements such as `class`, `property`, and `operation`. A class model would contain instances of those elements such as `ClassX` has `PropertyA` and `OperationB`, and is a subclass of `ClassA`. Finally, the class diagram would store the graphical location of each of the previous model's elements.

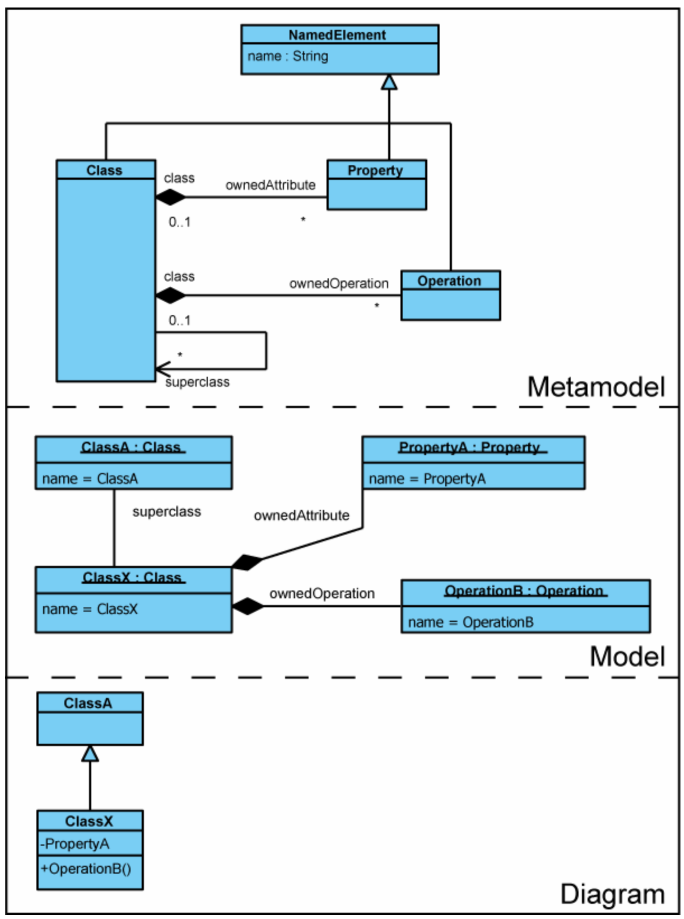


Figure 4. *Relationships between Metamodels, Models, and Diagrams*

7.1.4 Documents

The last repository of the OASIS functional architecture contains the documents associated with a software system under study, as the documentation about a legacy system can help in recovering and understanding its architecture. However, very often, the system's documentation is of poor quality and outdated [34]. In such cases, the Documentation Generation subsystem described in Section 7.8 attempts to derive the documentation from source code.

7.2 Data Access

The Data Access subsystem handles the mapping of low level data elements to higher level constructs. This supports the goal mentioned in section 7.1.2, i.e., to allow different tools to use their own data schema to persistently store information. The service groups of the present subsystem provide functionalities to define and transform data elements to conform to the data schema formats of the different tools to be integrated.

7.2.1 Data Object Definition

To allow different tools integrated within OASIS to interoperate, their data elements must be compatible. If they all use the same exchange model, such as the one described in section 7.3.1, data object definition consists of mapping it to the persistence technology used. For example, if a relational database is used to persistently store the data elements, a tool such as Hibernate [35] could be used to generate the corresponding database schema.

In the case where the tools to be integrated each have a different exchange model, the definition of data objects is more difficult. For example, some data elements can be stored in a relational database, while others are contained in an eXtensible Markup Language (XML) file. In such circumstances, transformation functions must be created to allow mapping between one format to another. An example of a tool which performs such transformation functions is Altova MapForce [36]. As illustrated in Figure 5 on the next page, it allows to visually define mappings between different formats such as XML and relational databases, and graphically create the required transformation functions.

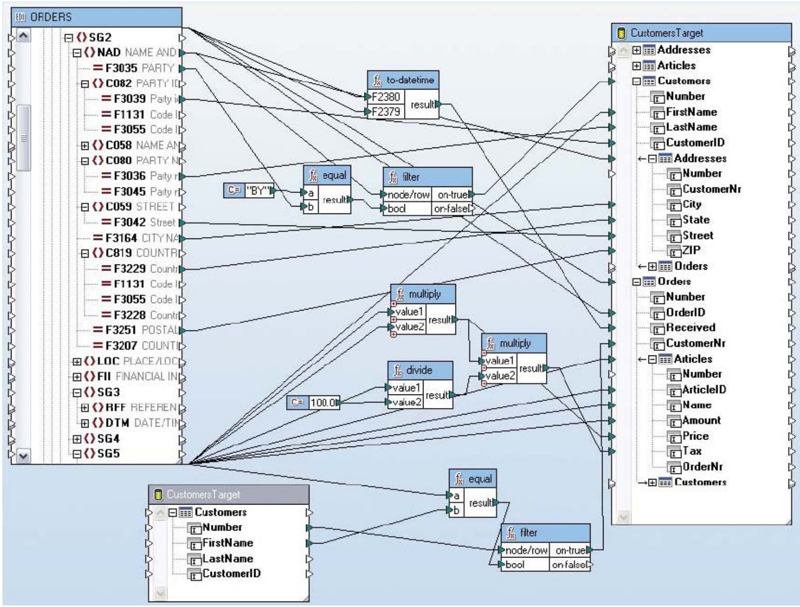


Figure 5. *Altova MapForce [36]*

To solve the problem of uniquely identifying information elements originating from different tools, the OASIS functional architecture uses the concept of Enterprise IDentifier (EID), as defined in [37]. Using this approach, each information element is given an identifier of the same size that is guaranteed to be unique. This uniqueness is achieved by giving each information producer a seed identifier which will form the prefix of the EID. When an information element is created, the producer typically concatenates its seed identifier with a sequence number to generate the EID. Figure 6 below, adapted from [37], illustrates the composition of 64-bit long EID.

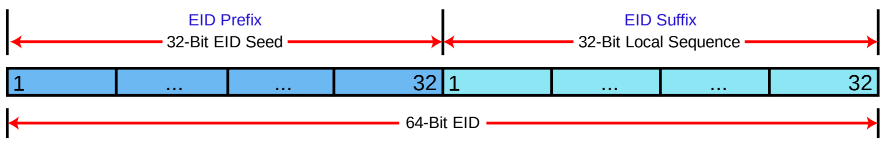


Figure 6. *Enterprise Identifier Composition*

This scheme supports up to 2^{32} (approximately 4.3 billion) information producers, which can each produce 2^{32} information elements.

7.2.2 Marshalling

Marshalling consists of pulling, from the repositories, information elements, packaging them into a data object, and then sending it to the information consumer that requested it. The implementation of this functionality is usually provided by the tool used for the definition of data objects. For example, Hibernate provides an API to load objects which have been persistently stored.

7.2.3 Unmarshalling

Unmarshalling is the opposite of marshalling. It consists of separating a data object into its constituent information elements and storing them in their corresponding repository. As for marshalling, the implementation of this functionality is provided by the data mapping technology used.

7.3 Information Management Services

Information Management Services are the top level subsystems of the OASIS functional architecture infrastructure. The others are the Repositories and Data Access, which were previously described in Section 7.1 and 7.2. Information Management mostly provides services to integrate a tool within OASIS and use the Repositories and Data Access subsystems.

7.3.1 Exchange Model Definition

Exchange models are the foundations for the other services provided by the functional architecture infrastructure. Before a tool can be integrated within OASIS, either by developing it or by reusing an existing one, its exchange model must be defined. For example, if a tool generating call graphs would have to be incorporated within OASIS, an exchange model similar to the one shown in Figure 7 below would first need to be defined.

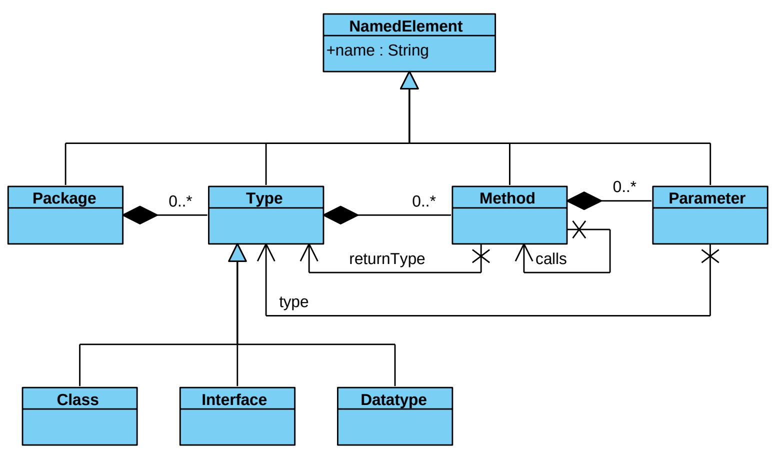


Figure 7. *Exchange Model for a Call Graph*

Once defined, the Data Object Definition service group could then generate its corresponding relational model and store it persistently into a database. This model could also be used by other subsystems, e.g., Visualization, which need to operate on call graphs. Furthermore, other information management services could retrieve and store instances of this exchange model without having to deal with the intricacies of the persistence mechanism used.

7.3.2 Browsing

Browsing consists of exploring a body of information, based on the organization of the collections or scanning lists, rather than by direct searching [38]. In the present case, browsing is performed by two kinds of actors: users trying to understand an unfamiliar software system and developers who want to extend the functionalities of OASIS.

The information contained in the repositories which is of most interest for the developers are the exchange models. When integrating a tool producing information in OASIS, a developer would first browse the available exchange models to see if there is one which already exists for the information provided by the tool. For example, the exchange model displayed in Figure 7 could be used by another tool producing call graphs for a different object-oriented programming language. For a developer integrating or developing a new information consumer, the browsing functionality would also allow to search for exchange models providing the information required by the tool.

In the case of users trying to understand an unfamiliar software system, they are rather interested in the actual instances of the information elements stored in the different repositories. This functionality can be seen as a basic tool to facilitate system understanding when there are no visualization technique available for certain types of information. Also, when several users are trying to understand the same system in a multi-user environment, browsing can be used to explore the information produced by others.

7.3.3 Querying

Querying is the process of retrieving information elements from a repository matching a set of criteria. An example of a criterion could be the method name of an instance of the exchange model displayed in Figure 7. Using pseudo-code, this criterion could be written as `callgraphmodel.method.name`, where `callgraphmodel` is the name of the call graph.

Multi-criteria querying is used to reduce the amount of information retrieved. A multi-criteria example could be the method and package name of an exchange model instance. Using pseudo-code again, these criteria could be expressed as `callgrahmodel.method.name` and `callgrahmodel.package.name`.

7.3.4 Publishing

Publishing is the main activity performed by information producers with respect to repositories. It consists of storing persistently the information contained in an instance of an exchange model. Upon publication, it becomes available to all the tools using the information contained in the repositories.

7.3.5 Subscription

The Subscription service can be seen as standing queries. Whenever information is stored in the repositories via the Publishing service, it is compared with user or tool specified criteria previously entered. If there is a match, it is forwarded, for example, to the user who requested it. The Subscription service allows to dynamically display or analyze information as it is produced. Also, in a multi-user environment, users can be notified of the information produced by others.

7.4 Fact Extraction

Fact extraction consists of finding pieces of information about a system. It is a fundamental step of reverse engineering and architecture recovery techniques and as a result, has often to be performed first [39]. This means that before any high-level reverse engineering analyses or architecture recovery activities can be performed, available information about a system has to be extracted and aggregated in a fact base. Such a fact base forms the foundation for further analysis tasks that are conducted next, either manually or (semi)-automatically using tools [39].

Fact extraction can either be static or dynamic. The functional architecture of OASIS supports both, as explained in the next two sections.

7.4.1 Static Fact Extraction

Static fact extraction provides information which is obtained by observing only the artifacts of a system [40]. A common technique for extracting static facts from source code is parsing.

7.4.1.1 Parsing

Informally, a parser is a program which receives input in the form of source code instructions and breaks them into parts such as objects, methods, and attributes [39]. This collected data, as well as the dependencies among the extracted entities, e.g., inheritance and association relationships, are then added to a fact base.

More formally, parsing transforms source code into a data structure, usually a tree, which is suitable for later processing and captures the implied hierarchy of the source code. A parser generally operates in a two-stage process. First, it identifies the tokens in the source code and then builds a parse tree using them.

A token is a categorized block of text, usually consisting of indivisible characters known as lexemes. Examples of tokens include literals, operators, and identifiers. A lexical analyzer initially reads the lexemes and categorizes them according to function, giving them meaning. This assignment of meaning is known as tokenization. A parse tree, or concrete syntax tree, is then generated from these tokens. A parse tree represents the syntactic structure of the source code according to a grammar.

In the context of OASIS, an abstract syntax tree (AST) is used instead of a parse tree. In a parser, an AST is an intermediate between a parse tree and a data structure. The latter is often used as a compiler or interpreter's internal representation of a computer program, while it is being optimized and from which code is generated.

An AST captures the essential structure of the source code in a tree form, while omitting unnecessary syntactic details. It differs from a parse tree by excluding nodes representing punctuation marks, such as the semi-colons terminating statements or the commas separating method arguments. It also omits tree nodes representing unary productions in the grammar. These omissions are represented by the structure of the AST [41].

Figure 8 below shows the AST of the Java source code displayed in Figure 9.

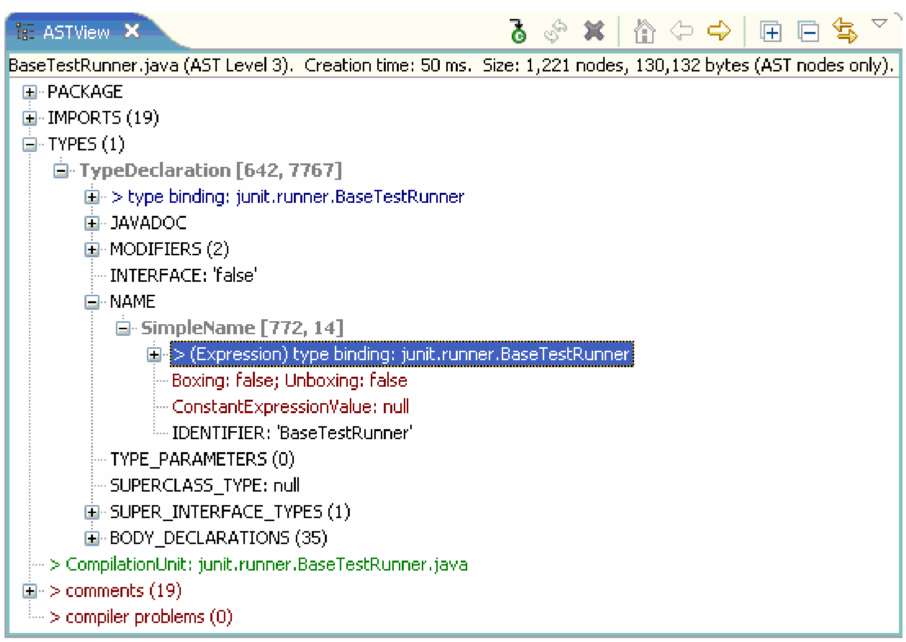


Figure 8. *View of an Abstract Syntax Tree*

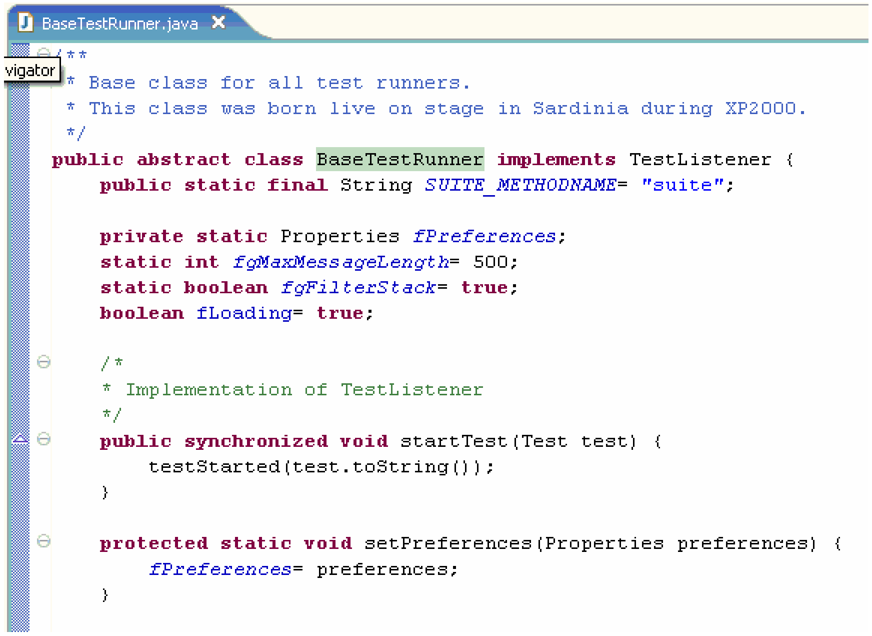


Figure 9. *Java Source Code*

7.4.1.2 Decompilation

In the cases where the source code of a system is not available, parsing cannot be performed. In these circumstances, the decompilation of the binary code has to be carried out instead.

A decompiler, or reverse compiler, is a program which attempts to perform the inverse process of the compiler. Given an executable program compiled using a high level programming language, the objective is to generate a high level language program which performs the same function as the executable program [42].

Decompiling executable programs is not a trivial task, as one faces several difficulties. The main problems are the separation of data and code, the reconstruction of control structures, and the recovery of high-level data types [43]. Also, any meaningful names given by programmers to variables and methods to facilitate their identification are not usually stored in an executable file. Therefore, they cannot be recovered by the decompiler. Another problem is the great number of subroutines introduced by the compiler [42]. To set up its environment and for runtime support, the compiler includes subroutines. These are usually written in assembler and most of the time, cannot be translated into a high level language. In addition, library routines, written either in the compiler language or in assembler, are also included by the linker. As an example, a "hello world" program compiled in C generates 23 different procedures [42]. To improve the decompilation process, decompilers make use of knowledge about certain compilers and libraries used in the compilation of the file to be decompiled [43].

One case for which the decompilation is somewhat easier is Java. The reason is that Java bytecode is relatively high-level and is guaranteed to be well-formed and well-typed due to verification constraints [44]. Therefore, it provides an ideal basis for decompilation back to Java source code. Another reason why decompiling bytecode is easier is that the most usual way of producing class files is to use Sun's javac compiler, which has specific compilation patterns [45]. However, decompiling Java bytecode has been complicated by the fact that there is an increasing number of compilers that can generate bytecode for other languages (e.g., AspectJ and C), as well as by the use of bytecode optimizers and obfuscators. These produce faster and/or smaller class files in the first case and classes which are hard to decompile and understand in the latter. Although the bytecode generated by these tools is both correct and verifiable, it is much more complex than the one produced by javac [44].

7.4.1.3 Build File Parsing

After a software system has been designed and implemented, it has to be configured, compiled, and linked for the particular environment in which it will be deployed [46]. For small systems developed for a unique platform, the make utility and a single Makefile, for example, are usually sufficient for system building. However, in the case of large and complex systems running on multiple platforms and supporting several functional configurations, the build process is more complicated.

Since the systems to be analyzed by the OASIS project will be large scale military applications, their configuration and built-time properties should be extracted from build management artifacts, such as build and configuration files [46]. Having the compilation dependencies between the compilation units of a system, the time-sequence configuration of the compilation procedure, as well as knowing which portions of source code are automatically generated at build time would provide valuable insights for the comprehension of an unfamiliar system.

The software systems to be analyzed will be military applications developed in C++ or Java. Therefore, examples of build file formats that should be supported would be Makefile [47] and Visual Studio Solution files for C++, and Ant files [48] in the case of Java.

7.4.1.4 Database Schema Extraction

Most software systems for business and industry are information systems, i.e., they maintain and process vast amounts of persistent data using database platforms [49]. This persistent data typically corresponds to the application domain elements whose values are relevant for the organization's business goals [50]. Data analysis is therefore a crucial step to identify the central business objects in a software system.

The objective of the Database Schema Extraction service group is to recover conceptual data models from physical databases. A conceptual model is expressed as an entity relationship (ER) schema, which consists of entity types, relationship types, attributes, as well as the various properties and constraints which translate the concepts and structures of the application domain [51]. This model should be structurally complete and semantically annotated. However, in most cases, important information about the data model is missing in the physical schema catalog extracted from the database [49]. Therefore, even though the Database Schema Extraction service group can provide support for extracting the schema catalog and reduce the effort spent in this phase, data analysis is a human-intensive activity and cannot be fully automated. It requires significant amount of experience and skills, as well as access to users and domain experts that can often contribute with valuable knowledge [49]. However, it has the potential to be a major aid in searching, collecting, and combining indicators for structural and semantic schema constraints, as well as to provide guidance from an initially incomplete data model to a complete and consistent result [49].

7.4.2 Dynamic Fact Extraction

Dynamic fact extraction provides information which is obtained by observing the system during execution [40]. As mentioned in Section 6, the heterogeneity and dynamism of today's software systems make it difficult to comprehend them outside the actual time and context in which they execute [52]. Therefore, architectural recovery cannot rely only on static information. It must be complemented by dynamic analysis, such as the exchange of control and data between the various components at run time. This information increases the level of precision provided by the static analysis and as a result, improves understanding. In general, when collecting dynamic information about a set of executions, one is interested in collecting information for some specific entities in the code (e.g., method calls and paths) and in a subset of the program (e.g., in a specific module or set of modules) [53].

7.4.2.1 Instrumenting

One technique commonly used to collect information about a system behavior is instrumentation. As opposed to general-purpose program transformations, instrumentation only aims to gather additional information about a system, rather than modify its original structure and behavior, allowing therefore only minor side effects, such as increases in execution time or changes to the log file [54]. As an example, Java bytecode instrumentation uses structural and semantic information provided by the language and platform specifications to both identify instrumentation points as well as avoid affecting the original program structure and behavior [54]. Such instrumentation does not remove program elements (e.g., classes, fields, and methods). Variables defined by the original program may be read but not written. Instrumentation may add its own variables, even to existing program elements (e.g., new fields or local variables), and those variables may be read or written by it. Instrumentation may also insert new code into original program methods, and invoke other methods from this code, provided that original variables are not modified as a result of these invocations. Finally, instrumentation may outline code, i.e., move all or part of the method code into a new method and replace it in the original method with the invocation of the new one [54].

Once executed, an instrumented program generates an execution trace, which can be defined as a record of the sequence of instructions executed that often takes the form of a list of code labels encountered [55].

There are two different kinds of instrumentation: source and binary. In the first case, trace statements are added into the source code of an application. In the second one, trace statements are inserted into binary code, which includes applications as well as dynamic and shared libraries. Instrumenting source code is easier than binary code, as one can work in a high-level language. However, the disadvantage is that after it has been instrumented, the modified source code has to be recompiled in order to be able to execute the tracing statements and therefore, extract dynamic information.

Due to the additional overhead for recompiling instrumented source code and the fact that the objective of the OASIS project is to recover and comprehend the architecture of large scale military software systems consisting of more than 1,000 classes, the present functional decomposition only considers binary instrumentation. The Instrumentation service group allows users to specify (1) the types of entities to instrument, (2) the parts of the code in which those entities must be instrumented, and (3) the kind of information to collect from the different entity types [56].

Figure 10 on the next page shows a sample Java program which prompts for a number and prints its factorial.

```
import java.io.*;

public class Factorial {
    private static BufferedReader in = new BufferedReader(new
        InputStreamReader(System.in));

    public static final int fac(int n) {
        return (n == 0)? 1 : n * fac(n - 1);
    }

    public static final int readInt() {
        int n = 4711;
        try {
            System.out.print("Please enter a number> ");
            n = Integer.parseInt(in.readLine());
        } catch(IOException e1)
             { System.err.println(e1); }
          catch(NumberFormatException e2)
             { System.err.println(e2); }
        return n;
    }

    public static void main(String[] argv) {
        int n = readInt();
        System.out.println("Factorial of "+  n + " is " +
        fac(n));
    }
}
```

Figure 10. *Java Source Code [57]*

Figure 11 displays the resulting bytecode when the above Java source code is compiled and Figure 12, the bytecode after it has been instrumented using the Byte Code Engineering Library (BCEL) [57].

```
0:  iload_0
1:  ifne           #8
4:  iconst_1
5:  goto           #16
8:  iload_0
9:  iload_0
10: iconst_1
11: isub
12: invokestatic   Factorial.fac (I)I (12)
15: imul
16: ireturn
```

Figure 11. *Java Bytecode [57]*

```
0:  sipush        4711
3:  istore_0
4:  getstatic     java.lang.System.out Ljava/io/PrintStream;
7:  ldc           "Please enter a number> "
9:  invokevirtual java.io.PrintStream.print (Ljava/lang/String;)V
12: getstatic     Factorial.in Ljava/io/BufferedReader;
15: invokevirtual java.io.BufferedReader.readLine ()Ljava/lang/String;
18: invokestatic  java.lang.Integer.parseInt (Ljava/lang/String;)I
21: istore_0
22: goto          #44
25: astore_1
26: getstatic     java.lang.System.err Ljava/io/PrintStream;
29: aload_1
30: invokevirtual java.io.PrintStream.println (Ljava/lang/Object;)V
33: goto          #44
36: astore_1
37: getstatic     java.lang.System.err Ljava/io/PrintStream;
40: aload_1
41: invokevirtual java.io.PrintStream.println (Ljava/lang/Object;)V
44: iload_0
45: ireturn
```

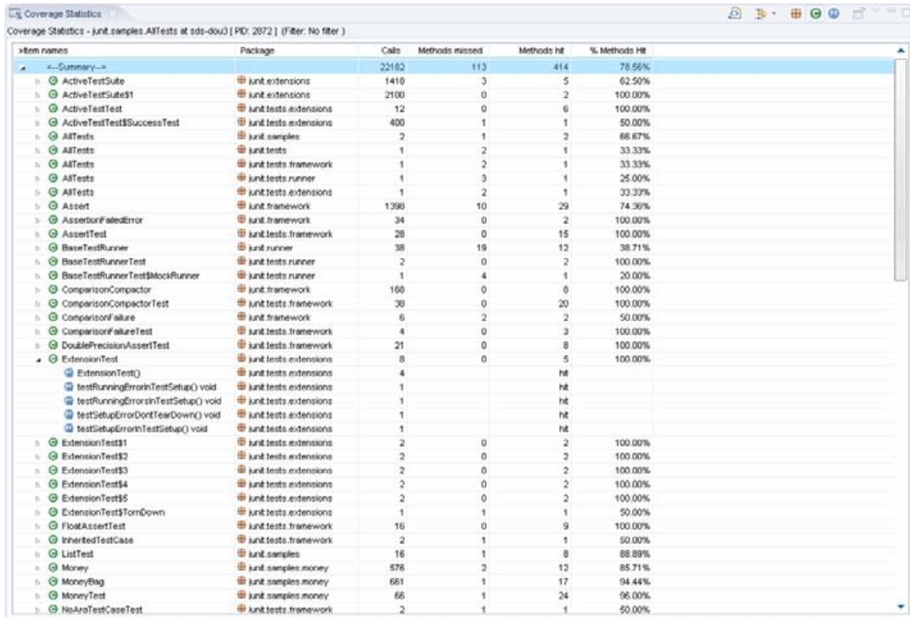
Figure 12. *Instrumented Java Bytecode [57]*

7.4.2.2 Profiling

Profiling injects instrumentation statements into the binary code of a software system to analyze the performance and resource utilization of its execution. It is useful for comprehension, as it allows identifying the portions of its source code which dominate execution time. It is also useful to get an understanding of the complex iterations between the source code, third-party libraries, operating system, hardware, networks, and other processes.

Figure 13 and 14 displayed next are two examples of profiling information that could be generated by the OASIS tool. They were produced using the Test and Performance Tools Platform (TPTP) Eclipse project [58].

Figure 13 shows the Coverage Statistics view of TPTP. It displays, for each class of the application, in this case, JUnit [59], its corresponding package, the number of times its methods were executed (Calls), the number of its methods which were executed (Methods hit) and not executed (Methods missed), as well as the ratio between the last two (% Methods hit).



Coverage Statistics

Coverage Statistics - junit.samples.AllTests at sds-dou3 [PID: 2872] (Filter: No filter)

»Item names	Package	Calls	Methods missed	Methods hit	% Methods Hit
<--Summary-->		22182	113	414	78.56%
ActiveTestSuite	junit.extensions	1410	3	5	62.50%
ActiveTestSuite$1	junit.extensions	2100	0	2	100.00%
ActiveTestTest	junit.tests.extensions	12	0	6	100.00%
ActiveTestTest$SuccessTest	junit.tests.extensions	400	1	1	50.00%
AllTests	junit.samples	2	1	2	66.67%
AllTests	junit.tests	1	2	1	33.33%
AllTests	junit.tests.framework	1	2	1	33.33%
AllTests	junit.tests.runner	1	3	1	25.00%
AllTests	junit.tests.extensions	1	2	1	33.33%
Assert	junit.framework	1398	10	29	74.36%
AssertionFailedError	junit.framework	34	0	2	100.00%
AssertTest	junit.tests.framework	28	0	15	100.00%
BaseTestRunner	junit.runner	38	19	12	38.71%
BaseTestRunnerTest	junit.tests.runner	2	0	2	100.00%
BaseTestRunnerTest$MockRunner	junit.tests.runner	1	4	1	20.00%
ComparisonCompactor	junit.framework	168	0	8	100.00%
ComparisonCompactorTest	junit.tests.framework	38	0	20	100.00%
ComparisonFailure	junit.framework	6	2	2	50.00%
ComparisonFailureTest	junit.tests.framework	4	0	3	100.00%
DoublePrecisionAssertTest	junit.tests.framework	21	0	8	100.00%
ExtensionTest	junit.tests.extensions	8	0	5	100.00%
ExtensionTest()	junit.tests.extensions	4		hit	
testRunningErrorInTestSetup() void	junit.tests.extensions	1		hit	
testRunningErrorsInTestSetup() void	junit.tests.extensions	1		hit	
testSetupErrorDontTearDown() void	junit.tests.extensions	1		hit	
testSetupErrorInTestSetup() void	junit.tests.extensions	1		hit	
ExtensionTest$1	junit.tests.extensions	2	0	2	100.00%
ExtensionTest$2	junit.tests.extensions	2	0	2	100.00%
ExtensionTest$3	junit.tests.extensions	2	0	2	100.00%
ExtensionTest$4	junit.tests.extensions	2	0	2	100.00%
ExtensionTest$5	junit.tests.extensions	2	0	2	100.00%
ExtensionTest$TornDown	junit.tests.extensions	1	1	1	50.00%
FloatAssertTest	junit.tests.framework	16	0	9	100.00%
InheritedTestCase	junit.tests.framework	2	1	1	50.00%
ListTest	junit.samples	16	1	8	88.89%
Money	junit.samples.money	578	2	12	85.71%
MoneyBag	junit.samples.money	661	1	17	94.44%
MoneyTest	junit.samples.money	66	1	24	96.00%
NoArgTestCaseTest	junit.tests.framework	2	1	1	50.00%

Figure 13. *TPTP Coverage Statistics View*

Figure 14 contains three pie charts generated by TPTP using the Business Intelligence and Reporting Tools (BIRT) charting library [60]. All three are at the package level and respectively list the top 10 packages in terms of base time, cumulative time, and number of calls, for a particular program execution.

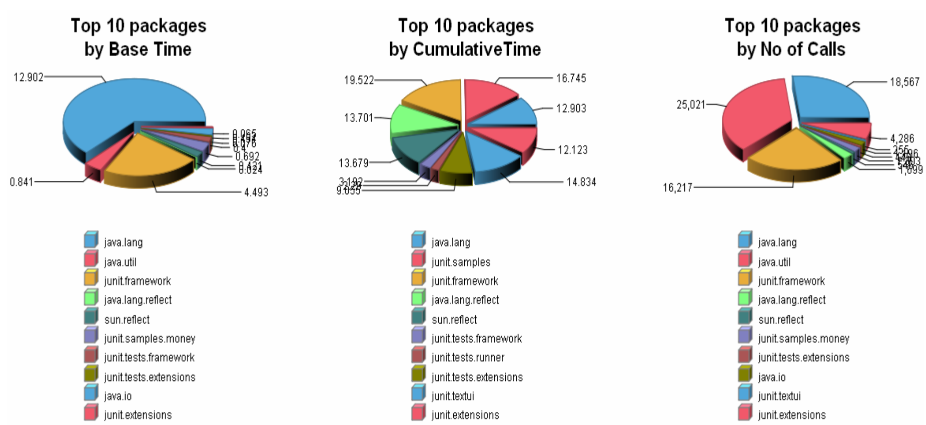


Figure 14. *TPTP Execution Information Report*

In the previous figure (Figure 14), the base time represents the time spent executing methods inside a package, excluding the time spent in other methods called by them. The cumulative time corresponds to the time the methods of a package spend on the execution stack, including both the time spent in the methods themselves and in the other methods they call. Finally, the number of calls counts the methods called by each package.

7.4.2.3 Monitoring

Like profiling, monitoring also instruments the binary code of a system. The objective is to observe its execution for the occurrence of specific events in real-time. Examples of events could be disk, database, or network access. Monitoring those events would further expand the knowledge about a system under study, as it could allow exploring its behavior and more precisely, the mechanisms it uses for interoperability.

Figure 15 displays a screen shot of the FileMon utility [61] from Microsoft. FileMon monitors and displays file system activity on a system in real-time. For the OASIS tool, functionality similar to this one should be implemented for file monitoring. For each request (e.g., open, read, write, or close), the responsible process as well as the path of the file being requested would be listed.

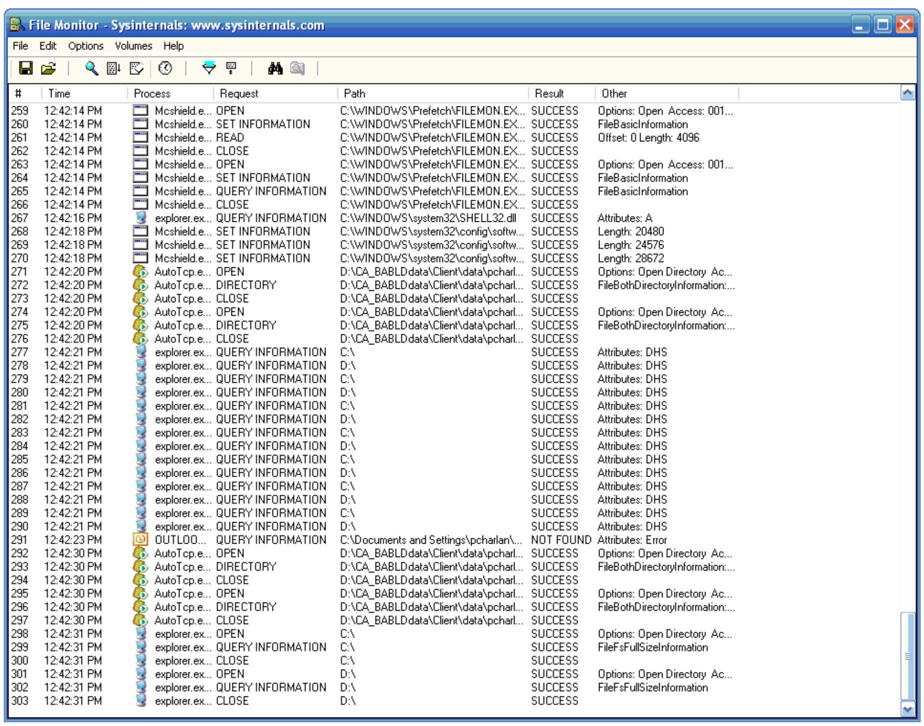


Figure 15. *FileMon for Windows*

7.5 Analysis

The Analysis subsystem relates to the separation of a system into its component parts. More specifically, it provides techniques to [62]:

- Separate a system into its constituent parts, in order to identify or classify the elements of communication.

- Make explicit the relationships among those elements to determine their connections and interactions.

- Recognize the organizational principles of the arrangement and structure that hold these elements together.

7.5.1 Software Metrics

A metric measures a property of a piece of software or its specifications. The OASIS functional architecture provides an extensive set of metrics, as it has been shown that they can provide guidance in analyzing the quality of the design and source code of a system, as well as its possible maintainability and comprehension [63]. The following list of metrics comes from [64], and each of them falls within one of the following three categories: size, complexity, or object-oriented.

Size metrics give an indication of the size of the source code, although thresholds to evaluate their meaning depend on the programming language and conventions used. In spite of this, size metrics should still be considered, as large classes and methods will be harder to understand than ones of a lesser size. Complexity metrics are usually considered as more meaningful measures of the size of a software system than simple size metrics, as they are not affected by the programming style. Object-oriented metrics can give an indication on the quality of the source code, especially from the viewpoint of classes and packages.

7.5.1.1 Size Metrics

Lines of Code (LOC). It is a size metric which simply counts the number of lines of code in a source file or module. It is generally agreed that LOC is not a very reliable metric, as it is affected by the coding style.

Non-Comment Non-Blank (NCNB). Also known as Source Lines of Code (SLOC), it is a more sophisticated version of LOC. It counts the number of lines of code, but excludes comments or blank lines.

7.5.1.2 Complexity Metric

Cyclomatic Complexity (CC). It is the most widely used complexity metric. It counts the number of linearly independent paths through a program module. It is calculated from the module’s control flow graph. $CC = e - n + 2$, where e is the number of edges in the graph and n, the number of nodes. Program modules with a high complexity tend to be more difficult to understand.

7.5.1.3 Object-Oriented Class Metrics

The following list of object-oriented class metrics comes from the work of Chidamber and Kemerer [65]. There is a significant amount of documentary evidence on the degree to which the next six metrics provide insight into the design process [64].

Weighted Methods per Class (WMC). It is the sum of the CC of all class methods. The number of methods and their complexity are predicators of the time and effort required to understand the class. Other information useful for comprehension that can be derived from this metric is that the larger the number of methods in a class, the greater is the potential impact on the classes inheriting them. Also, classes with a large number of methods are more likely to be application specific.

Response for a Class (RFC). Counts the number of methods that can be invoked as a result of a message sent to an object of the class or by some methods in the class. It also includes all the methods accessible within the class hierarchy. The larger the RFC, the greater is the complexity of the class and therefore, the more difficult it is to understand it.

Lack of Cohesion (LCOM). Counts the sets of methods in a class that are not related through the sharing of some of the class's instance variables. High cohesion indicates good class subdivision. A lack or low cohesion increases complexity and therefore, comprehension. Classes with low cohesion should be subdivided into two or more subclasses.

Coupling Between Object Classes (CBO). It is the number of distinct non-inheritance related class hierarchies on which a class depends. The higher the CBO of a class, the more difficult it is to understand it, as it is interrelated with other classes.

Depth of Inheritance Tree (DIT). Counts the number of ancestors of a class, from the class node to the root of the inheritance tree. The deeper a class is within the inheritance hierarchy, the greater the number of methods it is likely to inherit and therefore, the more complex is the prediction of its behavior.

Number of Children (NOC). Measures inheritance by counting the number of immediate subclasses of a class. It gives an indication of the possible influence a class has on the design of the system.

7.5.1.4 Object-Oriented Package Metrics

The following suite of object-oriented package metrics is based on the work undertaken by Martin [66]. For Java, the notion of a package is well defined. In the case of C++, it is defined as the set of classes in the modules of a single directory [64].

Afferent Coupling (Ca). Counts the number of other packages which depend on classes within the analyzed package. Ca is an indicator of the level of responsibility of a package.

Efferent Coupling (Ce). Counts the number of other packages that the classes within the analyzed package depend upon. Ce is an indicator of the package's independence.

Abstractness (A). It is the ratio of the number of abstract classes within a package relative to the total number of classes it contains. The range of this metric is from 0 to 1. An abstractness value of zero ($A = 0$) indicates a completely concrete package, while a value of one ($A = 1$) indicates a completely abstract package.

Instability (I). Instability is defined as the ratio between efferent and total coupling ($Ca + Ce$). This metric is an indicator of the package's resilience to change, i.e., the effort to change a package without impacting other packages within the application. The range of this metric goes from 0 to 1. An I of 0 reveals a completely stable package, while an I of 1 indicates that the package is unstable.

Distance from the Main Sequence (DMS). Calculates the perpendicular distance of a package from the idealized line given by $A + I = 1$. It indicates the package's balance between abstractness and stability. A package squarely on the main sequence is perfectly balanced with respect to abstractness and stability. Ideally, packages should either be completely abstract and stable ($x = 0, y = 1$), or completely concrete and unstable ($x = 1, y = 0$). The range for this metric goes from 0 to 1. A DMS of 0 indicates that a package is coincident with the main sequence, while a DMS of 1 reveals that the package is as far as possible from the main sequence.

7.5.2 Feature Location

A feature is defined as a behavior that is observable to a user interacting with a system [67]. The mapping between a feature and the source code implementing it is termed in software engineering as feature location [68].

There exist several techniques to assist with the task of feature location. The present functional architecture supports two of them. The first technique is software reconnaissance, while the second one makes use of concept analysis, a mathematical technique to investigate binary relations.

Software reconnaissance [68] is an automated feature location technique which uses dynamic analysis of scenario execution. A scenario is a sequence of user inputs triggering actions of a system which yields an observable result to an actor [23]. A scenario is said to execute a feature if the observable result is executed by the scenario's actions [69].

Software reconnaissance works as follows [69]:

1. The invoking input set *I*, i.e., a set of scenarios that will execute the feature, is identified.

2. The excluding input set *E*, i.e., a set of scenarios that will not execute the feature, is identified.

3. The system is executed twice using *I* and *E* separately.

4. By subtracting all the methods in the execution trace for *E* from those in the execution trace for *I*, the remaining methods are the ones that specifically deal with the feature and are a starting point for a more detailed static analysis.

The above approach for feature location deals with one feature at a time and gives little insight into connections between a set of related features. To derive detailed relationships between features and executed programs, a second technique [69] using concept analysis is proposed. Concept analysis is a mathematical technique that provides insights into binary relations [70]. The activities this second technique involves are briefly described next. For a more detailed description, please refer to Appendix A.

1. The set of relevant features *F* is identified.

2. The set of scenarios *A* is identified so that the features in *F* are covered.

3. Execution traces are generated so that all methods executed for each scenario are identified.

4. The relation table *R* between scenarios and methods is created.

5. Concept analysis is performed for the relation table *R*.

This technique identifies methods jointly required by any subset of features, classifies methods as low or high-level with respect to the given set of features, reveals additional dependencies between methods, and helps to identify the methods that together constitute a larger component [69].

7.5.3 Domain Knowledge Exploitation

One of the objectives of software comprehension is to understand the domain semantics of source code, i.e., to understand the functionality of the source code in terms of the system’s application domain. A domain is a problem area characterized by its vocabulary, common assumptions, architectural solution approaches, and literature [71]. As will be discussed in Section 7.9.3, the Domain Knowledge Definition service group of the present functional architecture allows users to capture and define the domain model of a software system. This knowledge will allow them, while analyzing the system under study, to map source code elements to their corresponding concepts of the application domain, as well as give them a set of expected constructs to look for in the source code. These could be computer representations of real world objects, algorithms, or overall architectural schemes.

7.6 Synthesis

According to Bloom's Taxonomy [62], synthesis consists of building a structure or pattern from diverse elements, i.e., to put parts together to a form a whole, with the emphasis on creating a new meaning or structure. The OASIS functional architecture provides three service groups to combine several source code elements to form a new whole at a higher level of abstraction: Design Pattern Recognition, Platform Model Transformations, and Clustering. These are explained next.

7.6.1 Design Pattern Recovery

One of the typical methods to understand the source code of a software system is to generate a graphical representation of its logical structure and behavior using, for example, the Unified Modeling Language (UML). The OASIS functional architecture supports the generation of such diagrams, as further described in Section 7.7. However, these graphical representations alone are often still insufficient for someone who is trying to completely understand a section of source code. One of the possible reasons, as stated by Beck and Johnson in [72], is that "existing design notations focus on communicating the 'what' of designs, but almost completely ignore the 'why'."

A design pattern "provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a given context" [73]. A design pattern not only contains a solution, i.e., the elements that make up the design, their relationships, responsibilities, but also the results and trade-offs of applying the pattern [74]. The recovery of design patterns helps in the understanding of a piece of source code, since a pattern provides knowledge about the role of each class within the pattern, the reason of certain relationships among pattern constituents and/or the remaining parts of a system. The recovery of design patterns is also important for comprehension, as they capture the rational behind the source code and can partially answer the question as to why is the system designed like that?

A typical system structure for design pattern recovery consists of three parts: a parser, detector, and database. Using the static facts extracted from the source code parser, the detector retrieves pattern definitions from the database, compares them with the extracted facts, and outputs the detection results [75]. Recovering design patterns is challenging and the precision of the currently existing techniques varies. Also, most of them can only identify a predefined set of structural design patterns.

7.6.2 Platform Models Transformations

The increasing number of technologies and target architectures that are available today for each component of a software system complicates the development of large scale systems. For example, each different platform results in different requirements for the design. As a consequence, the resulting designs are not portable, as they are too specifically related to a particular technology platform. Selecting a different platform, or simply changing the version of the platform used, necessitates great efforts.

As a solution to the above problem, the OMG has defined an approach to software system specification called Model Driven Architecture (MDA). MDA separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform [76]. This approach, as well as the standards which support it, allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms [76]. This is achieved by the use of Platform Independent Models (PIMs) and Platform Specific Models (PSMs).

A PIM is a model, expressed in UML, of a subsystem that contains no information specific to the platform or the technology that is used to realize it [77]. A PSM is a model of a subsystem that includes information about the specific technology that is used in its realization on a specific platform and hence, possibly contains elements that are specific to the platform [77]. This is also a UML model but expressed, because of the conversion step, in a dialect, i.e., a profile, of UML that precisely mirrors technical run-time elements of the target platform [78]. Note that the semantics of the PIM are also carried into the PSM.

An example of a PIM could be the formal definition of an operation that transfers funds from a checking to a savings account, specifying the amount to be subtracted from a designated checking to a designated savings account, as well as the constraint that the two accounts must belong to the same customer [76]. On the other hand, a specification of the funds transfer operation depending on interfaces to artifacts of CORBA, like the ORB, Object Services, or GIOP/IIOP would be a PSM.

In MDA, how the functionality specified in a PSM is derived from a PIM is done through transformations. As illustrated in Figure 16, when a PIM is sufficiently refined to be projected to the execution infrastructure, it is transformed into one or more PSMs, as each technology platform necessitates its specific PSM. Following this, the application code is generated. For component environments, it will consist of producing several types of code and configuration files, such as interface, component definition, program code, component configuration, and assembly configuration files [78].

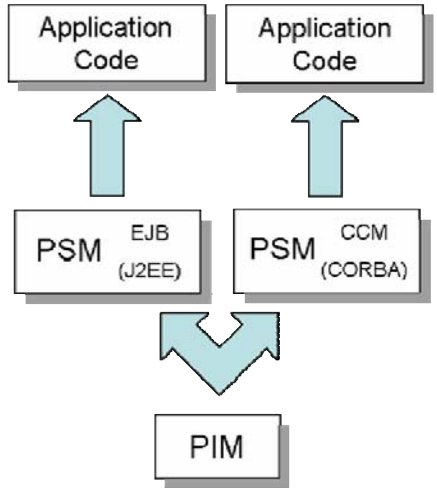


Figure 16. *MDA Transformations*

As illustrated in Figure 17, the Platform Models Transformations service group of the present functional architecture attempts to inverse the order of the MDA transformations. Starting from the application code, the PSM would be recovered. Then, it would be transformed to produce a complete PIM.

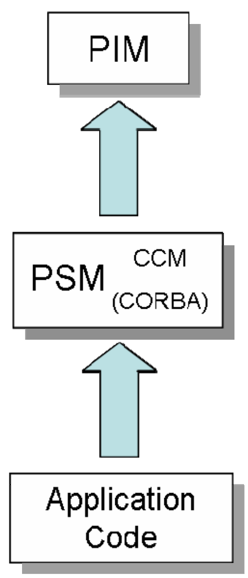


Figure 17. *Inverse MDA Transformations*

Recovering the fundamental precise structure and behavior of a software system in the PIM from implementation specific concerns contained in the PSM would facilitate its comprehension. It is easier to understand a model uncluttered by platform-specific semantics [76]. For example, a PSM need to use platform concepts of exception mechanisms, parameter types (e.g., platform-specific rules about objects references, value types, and semantics of call by value), as well as component model constructs, while a PIM does not need these characteristics and can use instead a simple and more uniform model [76]. By abstracting the implementation details, one can focus on the functionalities provided by a system and recover its domain model contained as part of the PSM.

7.6.3 Clustering

One way to understand a large scale software system is to decompose it into smaller subsystems which are more manageable and easier to understand. One technique which has been designed to facilitate the execution of this task automatically is software clustering. A cluster is commonly defined as a category of objects which exhibit similar features or properties [79]. Clustering aims at partitioning the source files of a software system into clusters, such that files which contain source code with similar functionality are placed in the same cluster, while files in different clusters contain source code that performs dissimilar functions [80]. Clustering techniques usually make use of criteria such as high cohesion and low coupling to decompose a software system into subsystems [81, 82, 83].

As mentioned in [79], in order to be useful for comprehension, an effective software clustering algorithm should propose clusters which follow familiar patterns and have appropriate names. The size of clusters should be kept at around 20 objects, by creating hierarchies of nested clusters. This approach, combined with effective visualization techniques, should produce clusterings that are useful for the comprehension of an unfamiliar system. An example of an algorithm which subscribes to this philosophy is the Algorithm for Comprehension-Driven Clustering (ACDC) presented in [79].

7.7 Visualization

The Visualization subsystem allows to generate modeling diagrams using, for instance, UML, to comprehend a system architecture from different views and using semantic zooming as a visualization technique. These views capture the decisions about the system's requirements, its logical and physical elements, as well as its structural and behavioral aspects. The different views generated by the present functional architecture are the design, interaction, implementation, and use case view. The different diagrams associated with each of them are indicated in Table 1 on the next page.

Table 1. *Diagrams Associated with Each View*

	Design View		Interaction View		Implementation View		Use Case View	
Diagram	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**
Package	*x*							
Class	*x*		*x*					
Component					*x*			
Interaction		*x*		*x*		*x*		*x*
Call Graph				*x*				
Statechart		*x*		*x*		*x*		*x*
Activity		*x*		*x*		*x*		*x*
Use Case							*x*	

7.7.1 Package Diagram

A package is a general-purpose mechanism for organizing elements (e.g., classes, interfaces, other packages) into groups [84]. It is graphically rendered as a tabbed folder. A package diagram is composed only of packages and the dependencies between them. Figure 18 below shows an example of a package diagram.

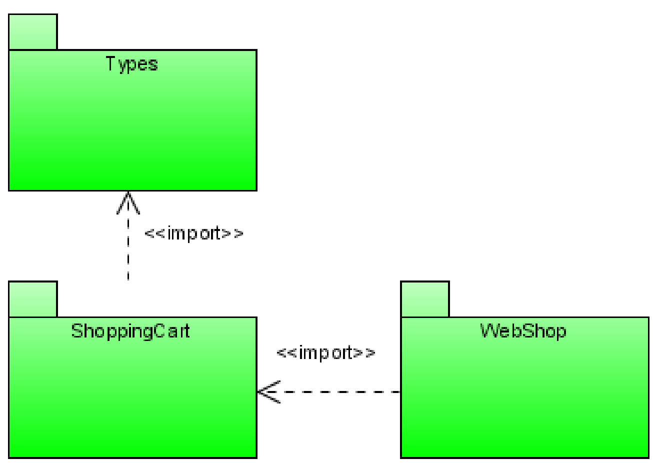


Figure 18. *Package Diagram [85]*

7.7.2 Class Diagram

Class diagrams model the static design view of a software system. They are used to visualize the static aspects of its building blocks, their relationships, as well as to specify their details for construction. As illustrated in Figure 19, a class diagram shows a set of classes, interfaces, collaborations, and their relationships [84].

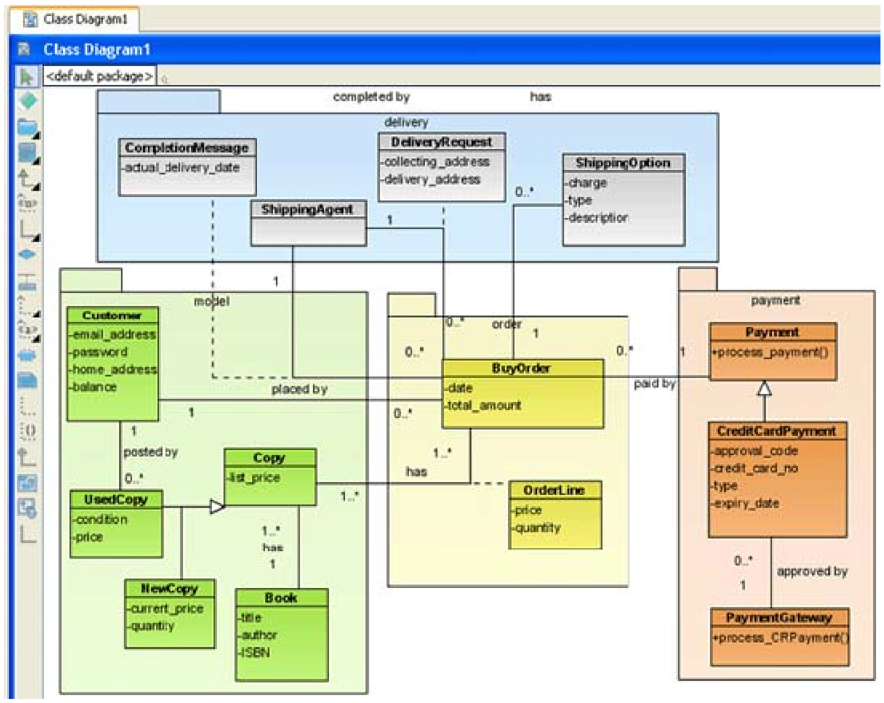


Figure 19. Class Diagram [85]

7.7.3 Component Diagram

UML components model the physical elements of a software system, such as executables, libraries, tables, files, and documents [84]. They typically represent the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs.

Figure 20 displays an example of a component diagram. It shows the organization and dependencies among a set of components. It addresses the static implementation view of a software system [84].

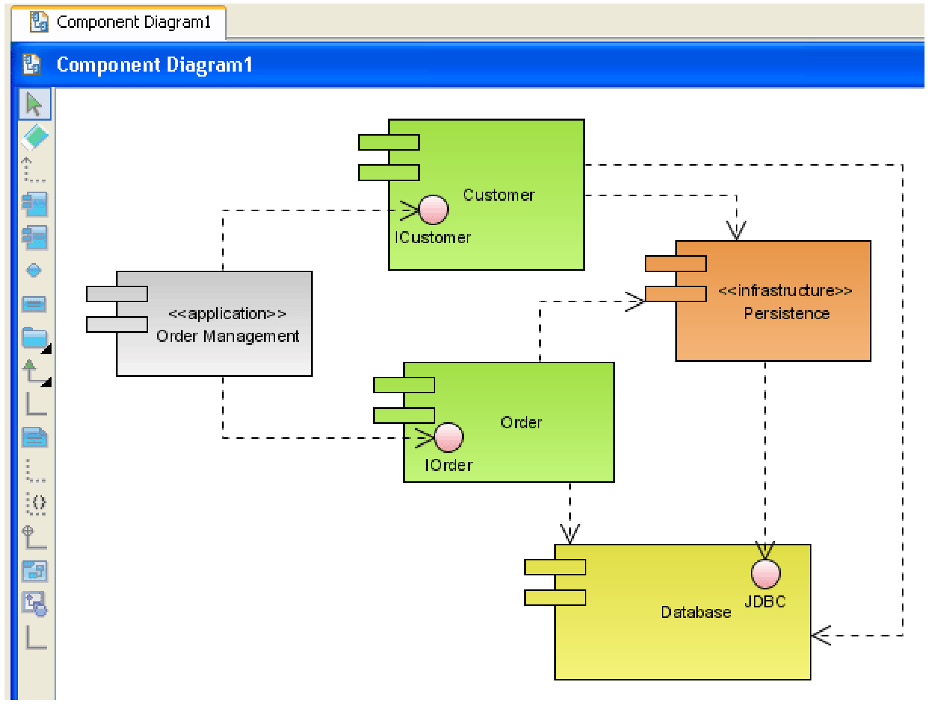


Figure 20. *Component Diagram [85]*

7.7.4 Interaction Diagram

Interaction diagrams model the dynamic aspect of a software system. They model concrete or prototypical instances of classes, interfaces, components, and nodes, along with the messages which are dispatched among them [84]. There are two types of interaction diagrams: sequence and communication diagram, the latter formerly known as collaboration diagram.

A sequence diagram, such as the one shown in Figure 21, emphasizes the time ordering of messages, while a communication diagram, like the one displayed in Figure 22, puts the accent on the structural organization of the objects which send and receive messages. Interaction and communication diagrams are semantically equivalent, i.e., one can be converted to the other without loss of information [84].

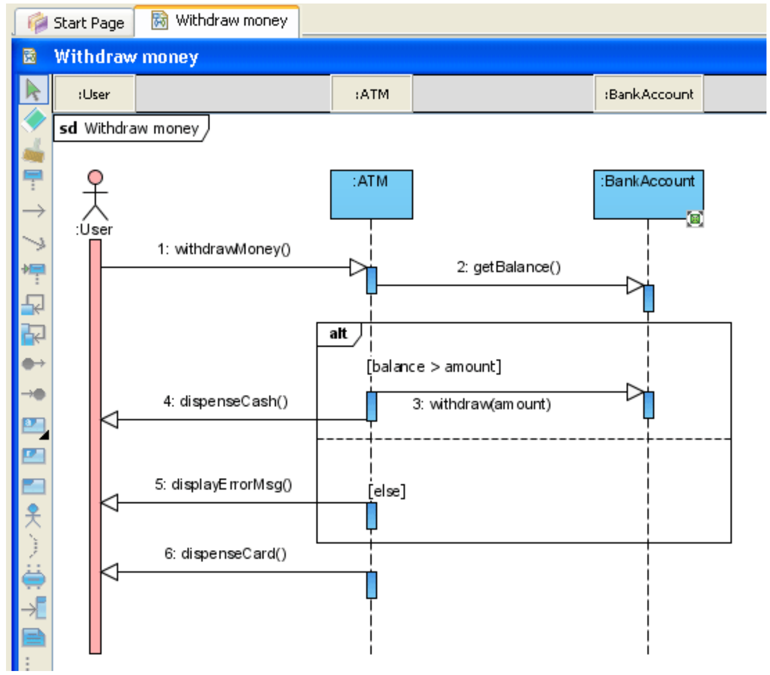


Figure 21. *Sequence Diagram [85]*

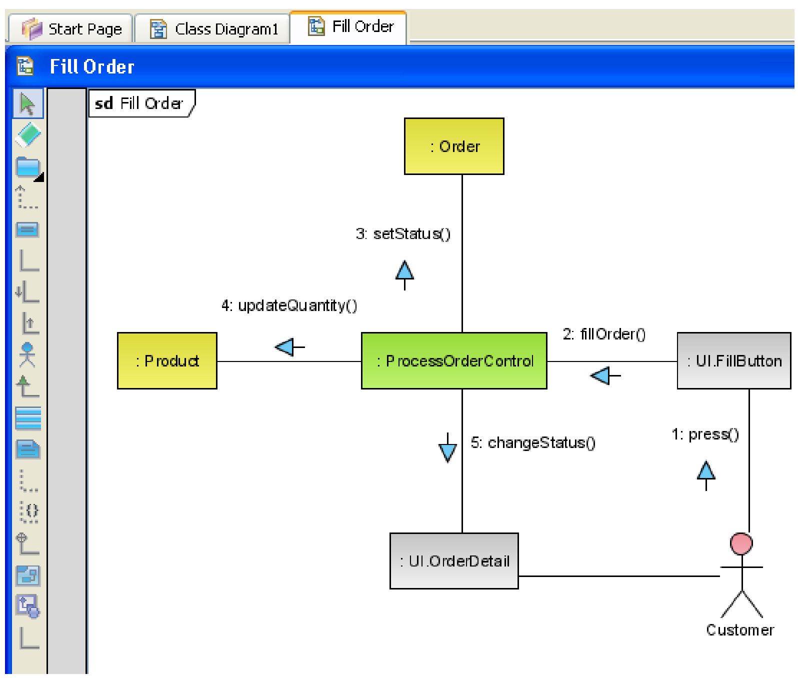


Figure 22. *Communication Diagram [85]*

7.7.5 Call Graph

A call graph is a directed graph which represents calling relationships between functions in a given program [86]. It shows the control flow of a program and can be determined partially using static analysis. In this case, it is usually regarded as a nondeterministic structure, as many branches are decided only at run time. A call graph can also be based on an execution trace, as illustrated in the Rational Quantify [87] call graph of Figure 23.

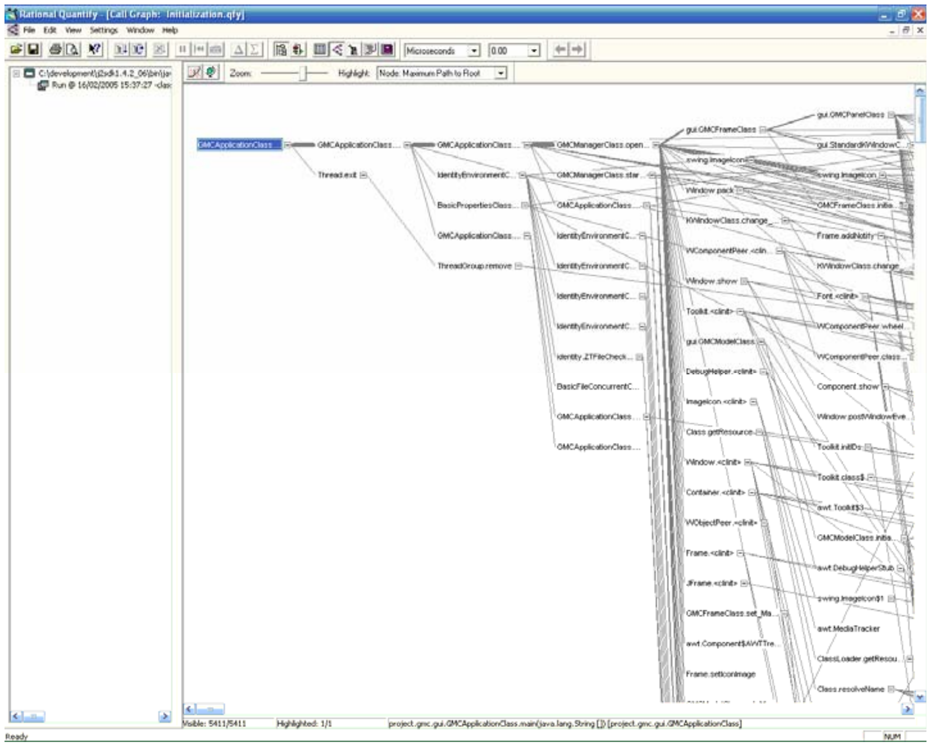


Figure 23. *Rational Quantify Call Graph*

7.7.6 Statechart Diagram

A state machine is a behavior which specifies the sequence of states a single object goes through during its lifetime in response to events, together with its responses to those events [84]. A state is represented as a rectangle with rounded corners. The relationships between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs is a transition [84]. It is rendered a solid directed line.

A statechart diagram is a diagram that shows a state machine, as illustrated in Figure 24.

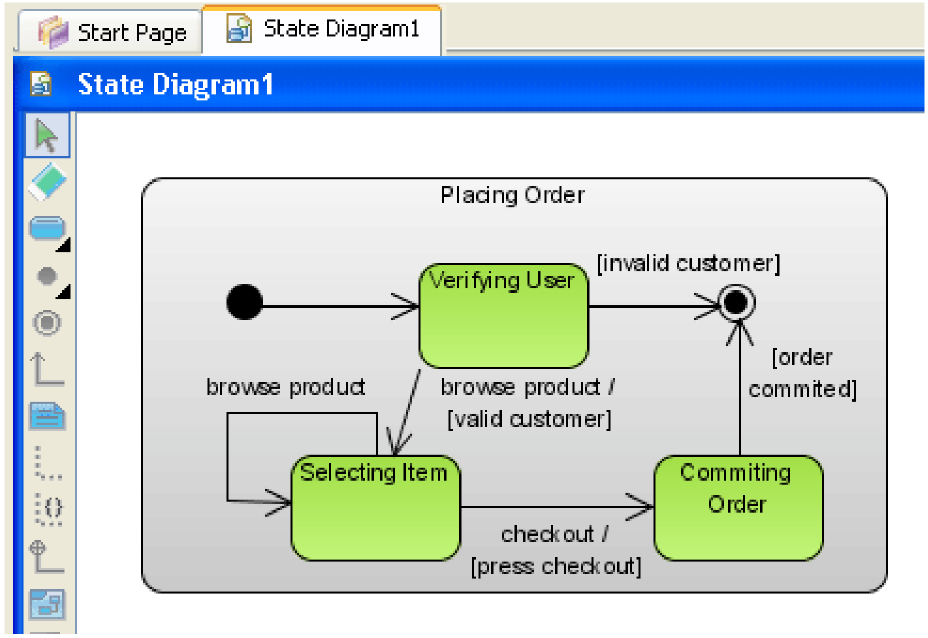


Figure 24. *Statechart Diagram [85]*

7.7.7 Activity Diagram

An activity diagram shows the flow from activity to activity. An activity is an ongoing nonatomic execution within a state machine. It ultimately results in some action, which is made up of executable atomic computations that result in a change of state in the system or the return of a value [84]. Calling an operation, sending a signal, creating or destroying an object, and evaluating an expression are examples of activities.

An activity diagram contains action and activity sates, transitions, branches, forks, and joins. Action and activity states are represented using a lozenge shape. A transition occurs when an action or activity of a state completes and the flow of control passes to the next one. It is represented as a directed line. A branch, depicted as a diamond, specifies alternate paths. Finally, a fork and join respectively represents the splitting of a single flow of control into two or more concurrent flows of control and their synchronization. Both are rendered as a thick horizontal line.

Activity diagrams are used to model a workflow or an operation. Figure 25 shows the activity diagram of a process order.

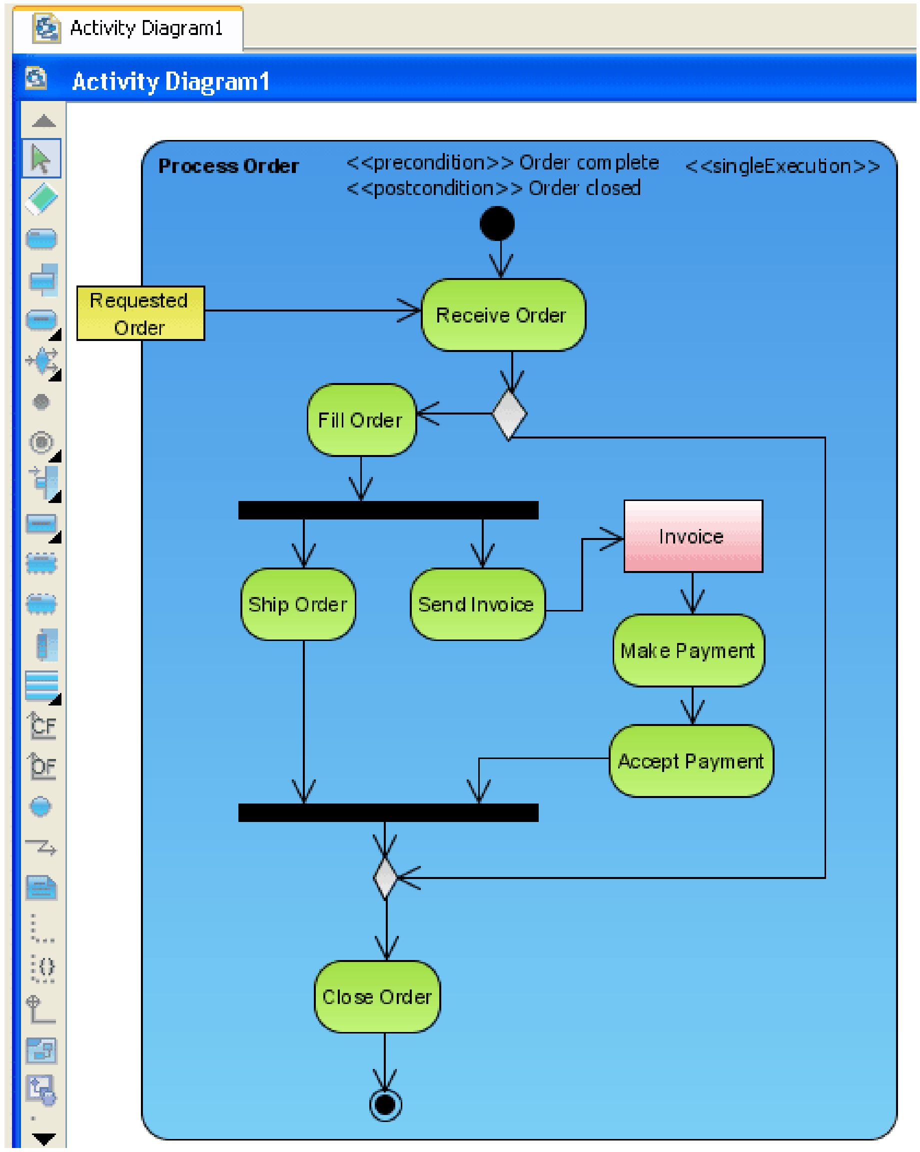


Figure 25. *Activity Diagram [85]*

7.7.8 Use Case Diagram

In UML, a use case specifies the behavior of a system, or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor [84]. It represents a functional requirement and is rendered graphically as an ellipse. An actor represents a coherent set of roles that users play when interacting with the system. Typically, an actor represents a role played by a human, a hardware device, or even another system [84]. Actors are rendered as stick figures.

A use case diagram shows a set of use cases, actors, as well as their relationships [84]. Figure 26 displays an example of a use case diagram for a sales order system.

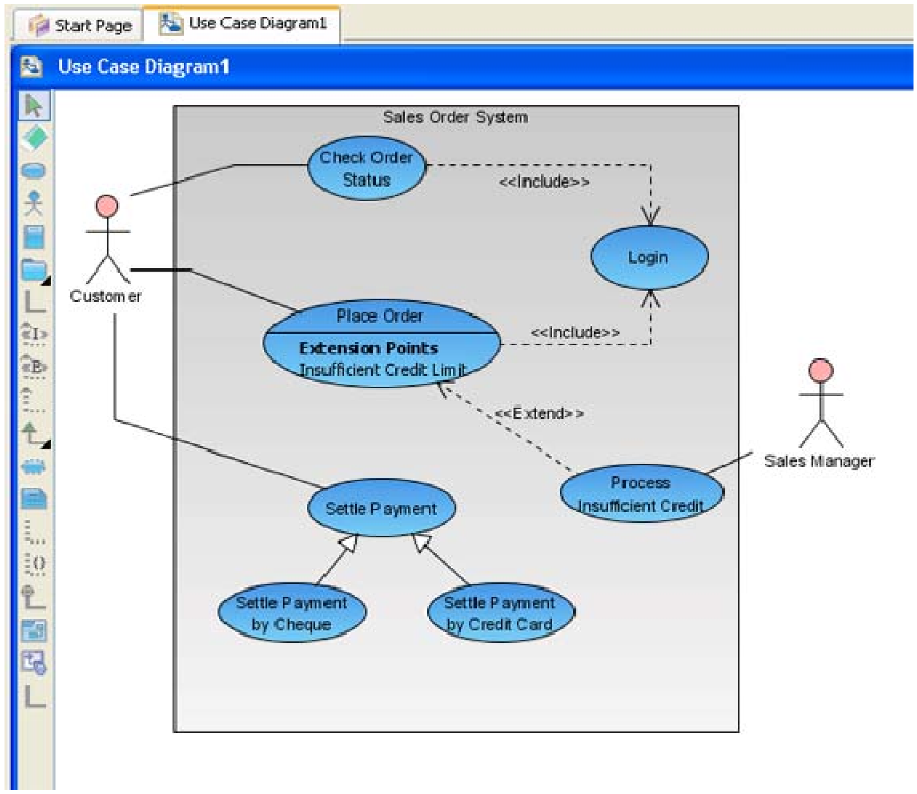


Figure 26. *Use Case Diagram [85]*

7.7.9 Semantic Zooming

Semantic zooming is a visualization technique defined by Bederson and Hollan [88]. The principle behind it is that as the viewpoint zooms on an area, the details not only become more distinct, as one would expect by virtue of being closer, but the representation also changes [89]. For example, in Figure 27, (a) displays the top level view of a method; (b) displays the top level view after semantic zooming has been triggered; (c) shows the resulting detailed view of the method, with the high level program elements of (a) still visible.

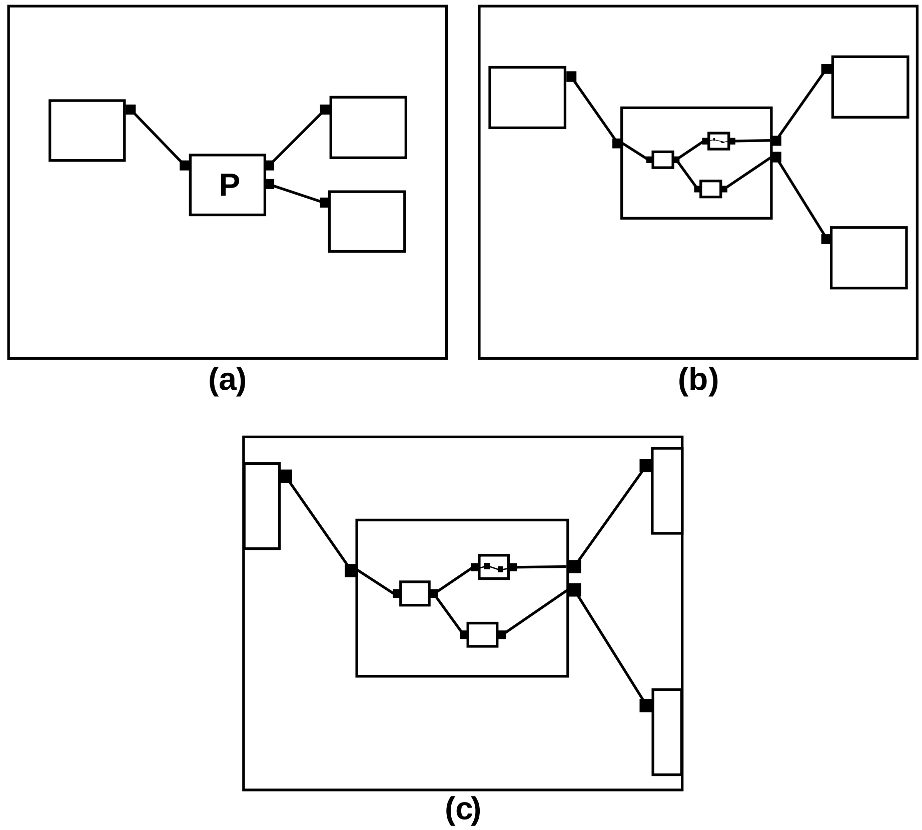


Figure 27. *Semantic Zooming*

In the present functional architecture, semantic zooming is used for package and class diagrams. When the viewpoint zooms on a package, the classes declared inside appear. If semantic zooming is still applied, the methods and instance variables of the class become visible.

7.8 Documentation Generation

It is commonly accepted that good documentation significantly improves the process of understanding a software system. However, as already mentioned, most existing systems have probably undergone several code revisions without a real concern about maintaining their documentation up to date [3]. One area of reverse engineering which intends to recover the documentation about an existing subject system is redocumentation [90]. Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level [90]. As shown in Figure 28, redocumentation cannot entirely be derived automatically. It has to be combined with information provided by hand. This information is supplied by users knowledgeable about the domain and/or system.

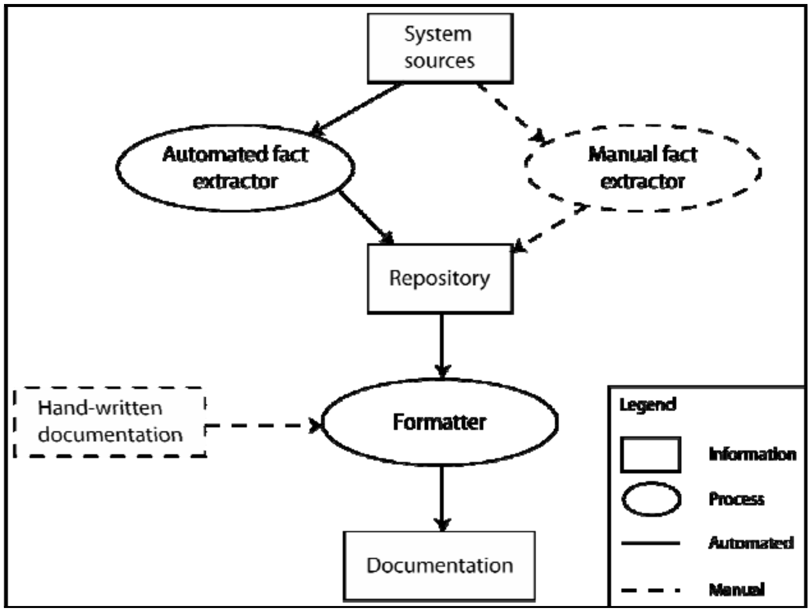


Figure 28. *Rocumentation Process [91]*

At the architectural level, the purpose of redocumentation is to recover the software system architectural description, which is organized into one or more constituents called views. Each view addresses one or more of the concerns of the system stakeholders and may consist of one or more architectural models [92]. Therefore, to redocument an architecture, multiple views and documents must be created. In addition, navigation mechanisms must be provided to assist in discovering related information among the various views and documents generated. Numerous documentation approaches make use of hypertext to link related documents [93, 94, 95].

The remaining of this section briefly describes three methodologies used in forward engineering and identifies some documentation artifacts they produce which could be recovered from an existing software system. However, as illustrated in Figure 29, there is a limit to what reverse engineering can recover.

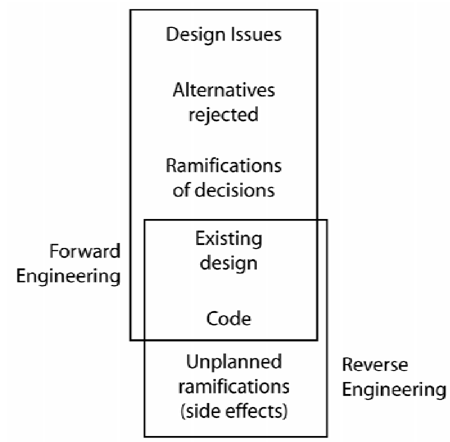


Figure 29. *Differences between Forward and Reverse Engineering Viewpoints [90]*

7.8.1 Rational Unified Process

The Rational Unified Process (RUP) is a software engineering process framework. It provides a disciplined approach to assign and manage tasks and responsibilities to produce high-quality software that meets the needs of its end users within a predictable schedule and budget [96]. RUP is based on a set of building blocks which describe what is to be produced, the necessary skills required, and the step-by-step explanation describing how specific development goals are achieved. The work products are probably the elements of most interest for documentation generation, as they represent the documents and models produced while working through the process. The major ones are illustrated in Figure 30.

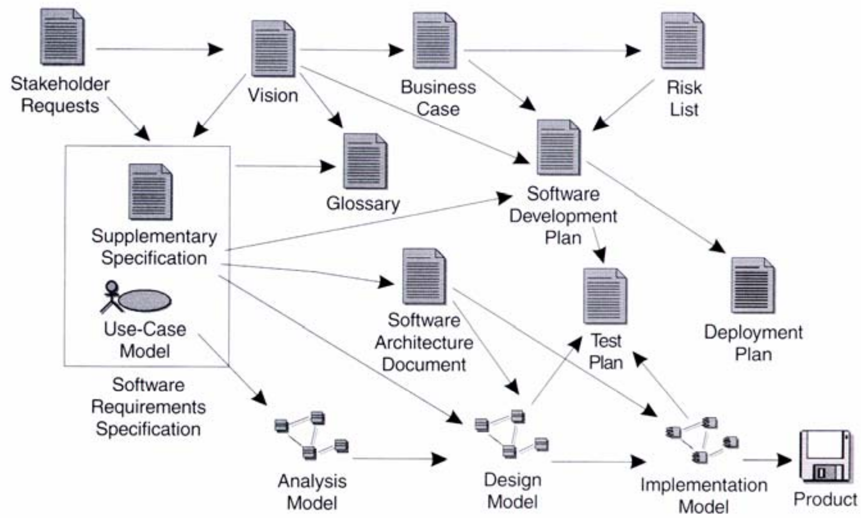


Figure 30. *Major Work Products of the Rational Unified Process [96]*

Only a subset of the above work products can be recreated from an existing software system. Examples are the analysis, design, and implementation model, which roughly correspond to the existing design and code elements of Figure 29. The software requirements specification contains use cases, which can also be retrieved using an approach such as the one described in [97]. The last work product from an existing software system that can partially be recovered is the deployment plan, which includes installation scripts and configuration documents. The remaining RUP work products cannot be recreated as they are too high level and too close to the application domain. It would require a considerable manual effort from a person knowledgeable about the system, its stakeholders, and the context in which it was developed.

7.8.2 Department of Defense Architecture Framework

The Department of Defense Architecture Framework (DoDAF) defines a common approach for DoD architecture description, development, presentation, and integration for both warfighting operations and business processes [98]. As illustrated in Figure 31, it is organized into four basic view sets: the overarching All View (AV), Operational View (OV), Systems View (SV), and Technical Standards View (TV). Each view has a number of products that describe its various aspects.

Although DoDAF is specifically suitable for large systems with complex integration and interoperability challenges, it does not represent software architectures. Some software architectural views are needed to supplement the DoDAF products to understand how well these systems will operate [99]. As a result, it is not a suitable candidate for documentation generation. However, in spite of that, some of the DoDAF products could be recreated from an existing system. One example is the TV-1 Technical Standards Profile, which documents the standards used in a given architecture. In the SV, the following products could also be recovered: the SV-4 Systems Functionality Description, which details the functions performed by systems and the information flow among system functions; the SV-10b Systems State Transition Description, that illustrates the responses of a system to events; the SV-10c Systems Event-Trace Description, which refines the critical sequences of events described in the OV; and the SV-11 Physical Schema, which is a physical implementation of the information of the OV-7 Logical Data Model.

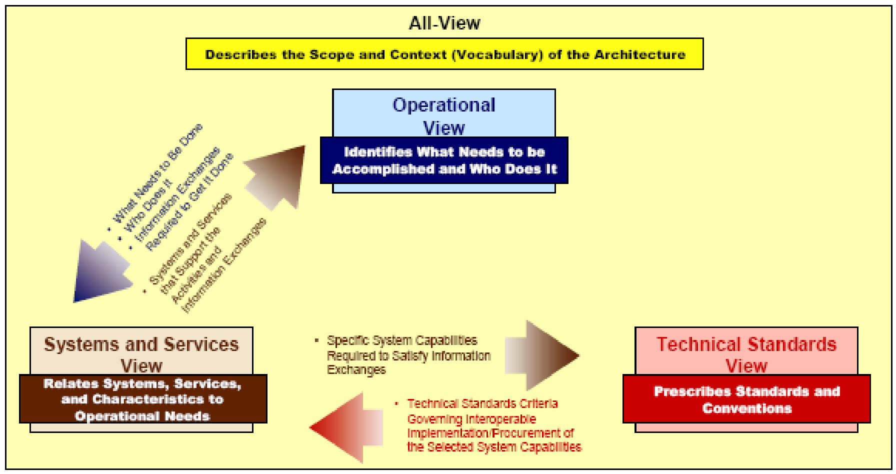


Figure 31. *DoDAF View Sets [98]*

7.8.3 IEEE 12207

The IEEE 12207 standard establishes a common framework for software life cycle processes [100]. As shown in Figure 32, these are defined in three broad categories: Primary Life Cycle Processes, Supporting Life Cycle Processes, and Organizational Life Cycle Processes. Each process is defined in terms of activities, which are in turn broken down into a set of tasks.

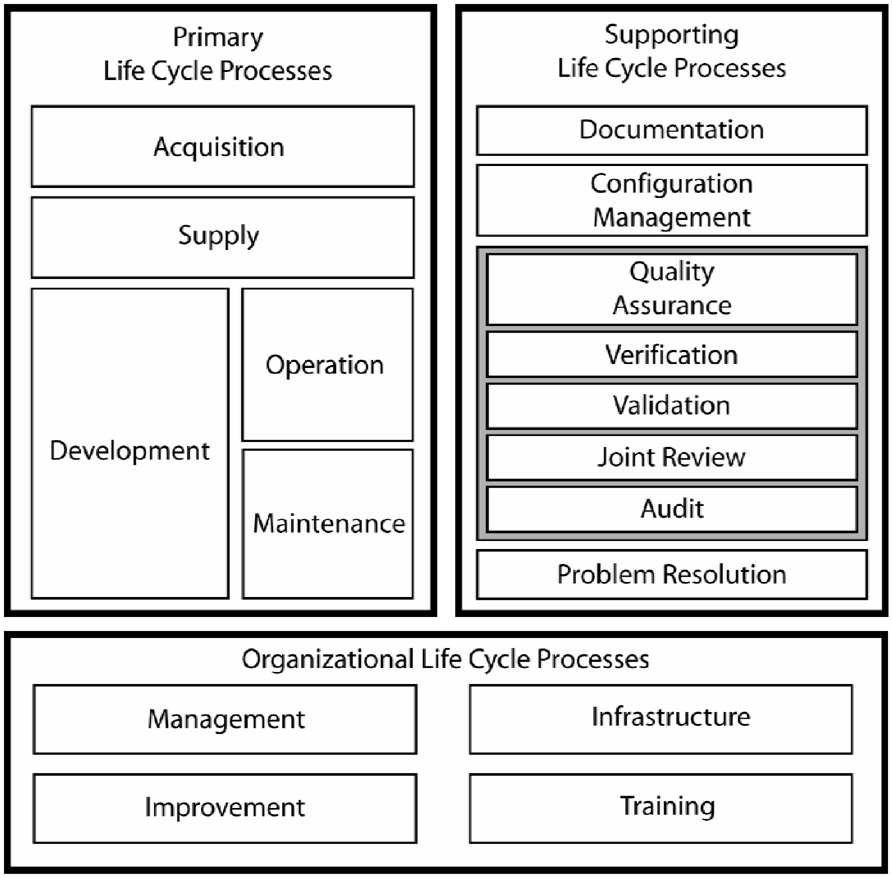


Figure 32. *12207 Life Cycle Processes [100]*

The IEEE 12207 standard is especially appropriate for acquisitions, as it recognizes the distinct roles of acquirer and supplier. In fact, it is intended for two-party use, where an agreement or contract defines the development, maintenance, or operation of a software system [101]. The set of processes, activities and tasks are then adapted to the software project. As the IEEE 12207 standard is a relatively high-level document, it does not specify the details of how to perform the activities and tasks comprising the processes, nor does it prescribe the format or content of documentation [101]. As a result, it may be difficult to redocument an existing software system. However, some information contained in the documents created as part of the development process activities could be recovered. These development activities, illustrated in Figure 33, are closely related to the major work products of the RUP described in Section 7.8.1.

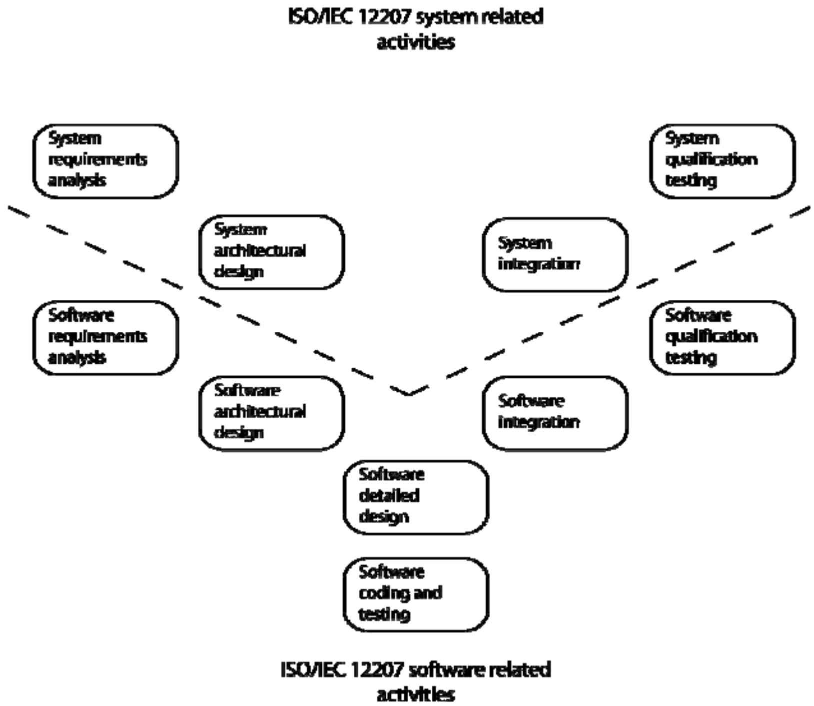


Figure 33. *12207 Activity Classification [102]*

7.9 Comprehension Process

Once implemented, the present functional architecture will expose potential users to a lot of information, as software systems to be comprehended will become larger and grow into complexity. Depending on the specific task a user wants to accomplish, some of the information and techniques provided by the OASIS tool could be irrelevant, therefore overloading the user with unrelated information. To prevent such situations from happening, the architecture recovery and comprehension tool should provide users with the necessary guidance in choosing the tools, abstraction levels, and analysis techniques that are appropriate for the specific task to be performed. This support will be provided through the Comprehension Process subsystem. This subsystem combines user experience with techniques and tools to reduce the complexity of the comprehension process, by providing only a coherent set of applicable tools and information relevant to the specific comprehension task.

7.9.1 Process Management

As mentioned in Section 3, there are many individual characteristics that will impact how a programmer tackles a comprehension task [7]. The most important ones are the program under study, the characteristics of the programmer, as well as the comprehension task to achieve. However, the currently existing tools do not take these characteristics into consideration, as they provide a one tool fits all approach. They do not distinguish between the different users' experience and knowledge levels, as well as the particular comprehension goals and tasks.

The OASIS functional architecture attempts to address this limitation by offering the possibility to create user specific architecture recovery and comprehension workflow templates. These templates are created by users to take into consideration their technical expertise, the comprehension tasks they want to complete, as well as the type of software system to be analyzed.

7.9.2 Process Execution

Once created, the architecture recovery and comprehension workflow templates can be instantiated to guide users while performing comprehension tasks. These could be similar to the cheat sheets provided by Eclipse. An example of an Eclipse cheat sheet is illustrated in Figure 34. A cheat sheet guides the user through a series of complex tasks to achieve an overall goal. Some steps can be performed by the cheat sheet and some are described so that the user can manually complete them.

The instantiated workflows would guide users through predefined tasks by providing them with a choice of tools, abstraction levels, and analysis techniques that might be useful during a specific task. The objective is to avoid exposing irrelevant information to the user.

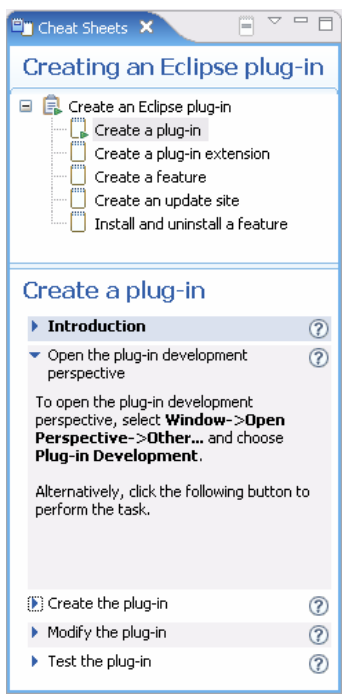


Figure 34. *Eclipse Cheat Sheet*

7.9.3 Domain Knowledge Definition

In Section 2.2, one of the cognitive models presented to understand a software system is the top-down approach. Using this strategy, a programmer reconstructs knowledge about the application domain and then maps it on the source code to identify the relevant software artifacts. However, one limitation of the current tools which was identified in Section 6 is that they do not offer functionalities to incorporate domain and user knowledge about a software system. This might prevent programmers from using the top-down approach as a comprehension strategy.

To overcome this limitation, the Domain Knowledge Definition service group allows users to capture and define the domain model of a software system to comprehend. This results in a vocabulary of terms representing entities of the domain and their relationships, which together imply certain semantic information. For example, if a user wanted to understand the Collaborative Operations Planning System (COPlanS) [103], an application developed by the Decision Support Systems Section at DRDC Valcartier to support the Military Operational Planning Process, then the user would use the Domain Knowledge Definition service group to define in a model, the elements of the application domain. In this particular case, such elements could be course of action, mission, operation, and risk. This model would then be used to map the elements of the application domain to their corresponding source code artifacts.

7.10 Graphical User Interface

The OASIS functional architecture aims at providing an environment into which comprehension tools can be integrated. This integration is supported at the data level by the infrastructure, which consists of the Information Management Services, Data Access, and Repositories subsystems. However, this integration must also be supported at the user interface level. This separation of concerns between the data and user interface is known as the Model-View-Controller architectural pattern [104]. In this pattern, the model represents the information on which the application operates, the view renders the model in a form suitable for interaction, and the controller processes and responds to user actions to invoke changes on the model. The controller can control several views. This is required to support the concept of multiple coordinated views, which enhance comprehension activities [105].

The coordination mechanism in the present functional architecture is achieved through the use of pluggable units of functionalities called plug-ins. This is similar to what is used in Eclipse. A plug-in provides functionality by hooking into extension points defined by other plug-ins. It can also define new extension points. Using this mechanism, several views can be coordinated. As an example, in Figure 35, when the `ListTest` item is selected in the `Package Explorer` view, the same element is selected in the `Outline` and `Properties` views and displayed in the editor. Also, selecting an item can cause the editor to scroll if this element was not visible.

Another concept present in Eclipse is user action. Plug-ins can add actions to menus, contextual menus, and toolbars, for example, and reuse existing actions. This helps in providing a consistent user experience across various integrated tools.

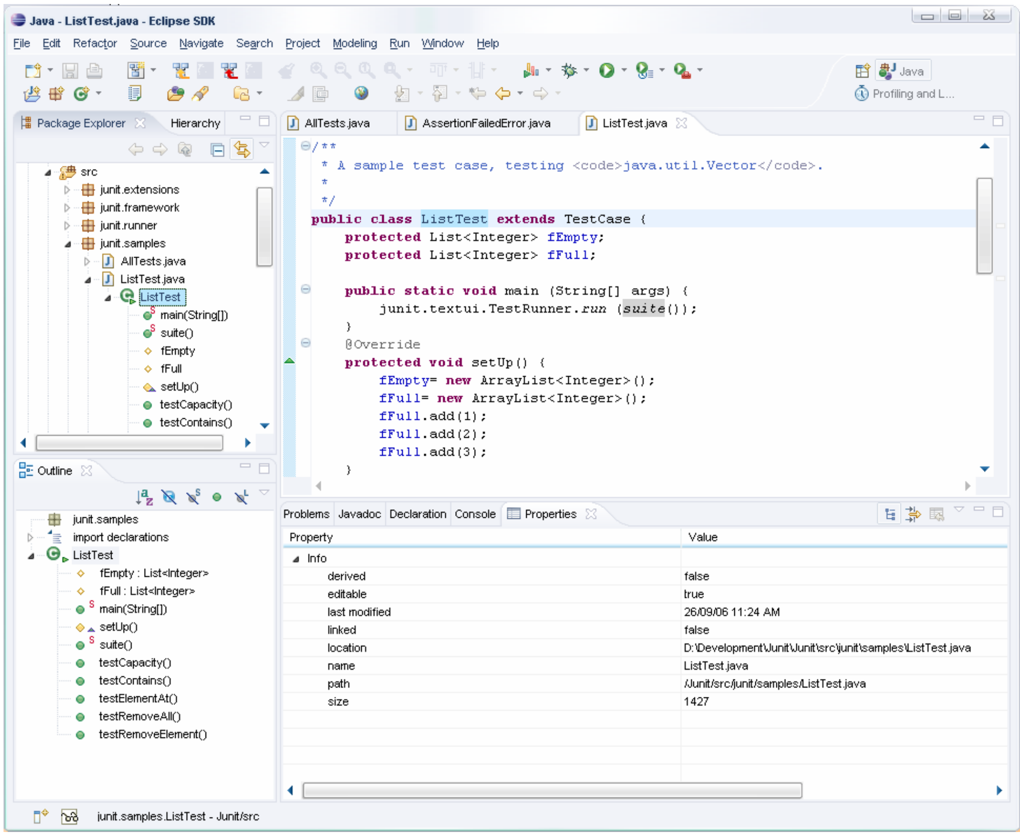


Figure 35. *View Coordination in Eclipse*

8. Conclusions and Future Work

The research community in the field of software comprehension has produced many diverse tools and prototypes to assist with the understanding of already existing systems [7]. However, the majority of these tools have not yet been adopted in the industry. One possible explanation for this is that the value of many research ideas has not been adequately substantiated through studies [11].

The present technical memorandum describes the functional decomposition of an architecture recovery and comprehension tool. This functional decomposition is based on the results of a qualitative study conducted by the OASIS research group using static and dynamic analysis tools to recover and comprehend the architecture of large scale military applications written in C++ and Java. It is also based on the findings contained in a state-of-the-art survey on system architecture recovery and comprehension. This functional decomposition is a way to synthesize the knowledge of the OASIS group in this research area. It serves as a reference model which is destined to evolve with the advancement of the group's knowledge in the area of architecture recovery and comprehension and orient its future work.

Following the conceptualization of this functional decomposition, the next step will consist of implementing a subset of it as a collection of Eclipse plug-ins to recover and comprehend the architecture of Java software systems. As already mentioned, Eclipse is an extensible open source IDE. Using Eclipse and its plug-in architecture will prevent the OASIS project from making the same mistake many existing tools have made, i.e., provide a suite of tools that aim for "one tool fits all." There already exist a wide variety of tools available as Eclipse plug-ins that can assist with program comprehension. The extensible architecture of Eclipse will allow the OASIS project to take advantage of these tools and integrate them with the technical solutions to be developed at DRDC Valcartier. Using Eclipse as a development framework will also allow the OASIS project to reuse the browsing and searching functionalities it offers and which were found to be useful for the comprehension of software systems in [6].

Ideally, once this prototype is developed, another study, similar to the previous one but with an improved design and set of comprehension tasks, should be conducted. Its objective would be to assess the added value of the OASIS architecture recovery and comprehension prototype on the understanding of participants. Future work should also consist of extending the prototype to support the architecture recovery and comprehension of C++ legacy systems, through the Eclipse C/C++ Development Tooling (CDT) project [106]. This would address one limitation of most existing tools, i.e., multi-language support.

9. References

1. The Technical Cooperation Program - Joint Systems and Analysis Group, "The Engineering and Acquisition of Systems of Systems in the United States DoD," *Tech. Report TR-JSA-TP4-1-2001*, Jan. 2001.

2. D. Garlan and D.E. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Trans. on Software Eng.*, vol. 21, no. 4, Apr. 1995, pp. 269-274.

3. R. Richardson, et al., "A Survey of Research into Legacy System Migration," *Tech. Report TCD-CS-1997-01*, Trinity College Dublin, Dublin, Ireland, Jan. 1997.

4. M. Lizotte and J. Rilling, "OASIS: Opening-up Architecture of Software-Intensive Systems", *Proc. of the 24th Army Science Conf. (ASC'04)*, Orlando, Fla., Nov. 2004.

5. J. Rilling, "State of the Art Report: System Architecture Recovery and Comprehension," *Tech. Report*, DRDC Valcartier, Val-Bélair, Que., 2003.

6. P. Charland, et al., "Using Software Analysis Tools to Understand Military Applications: A Qualitative Study," *Tech. Memorandum TM 2005-425*, DRDC Valcartier, Val-Bélair, Que., 2005.

7. M.-A.D. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," *Proc. of the 13th Int'l Workshop on Program Comprehension (IWPC'05)*, St. Louis, Mo., May 2005, pp. 181-191.

8. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. on Software Eng*., vol. 10, no. 5, Sept. 1984, pp. 595-609.

9. E. Soloway, et al., "Designing Documentation to Compensate for Delocalized Plans," *Comm. of the ACM*, vol. 31, no. 11, Nov. 1988, pp. 1259-1267.

10. R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l J. of Man-Machine Studies*, vol. 18, no. 6, June 1983, pp. 543-554.

11. M.-A.D. Storey, K. Wong, and H.A. Müller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?," *J. Science of Computer Programming*, vol. 36, no. 2-3, Mar. 2000, pp. 183-207.

12. B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *Int'l J. of Computer and Information Sciences*, vol. 8, no. 3, June 1979, pp. 219-238.

13. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.

14. N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, 1987, pp. 295-341.

15. S. Letovsky, "Cognitive Processes in Program Comprehension," *Proc. of the 1st Workshop on Empirical Studies of Programmers*, *Ablex Publishing*, 1986, pp. 58-79.

16. D.C. Littman, et al., "Mental Models and Software Maintenance," *1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, Washington, D.C., 1986, pp. 80-98.

17. A. von Mayrhauser and A.M. Vans, "From Code Understanding Needs to Reverse Engineering Tool Capabilities," *Proc. of the 6th Int'l Workshop Computer-Aided Software Eng. (CASE'93)*, Singapore, Jul. 1993, pp. 230-239.

18. M.-A.D. Storey, F.D. Fracchia, and H.A. Mueller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization," *Proc. of the 5th Int'l Workshop on Program Comprehension (IWPC'97)*, Dearborn, Mich., May 1997, pp. 17-28.

19. S.R. Tilley, S. Paul, D.B. Smith, "Towards a Framework for Program Understanding," *Proc. of the 4th Int'l Workshop on Program Comprehension (IWPC'96)*, Berlin, Germany, Mar. 1996, pp. 19-28.

20. F. Détienne, *Software Design: Cognitive Aspects*, Springer, 2001.

21. I. Vessey, "Expertise in Debugging Computer Programs: A Process Analysis," *Int'l J. of Man-Machine Studies*, vol. 23, no. 5, Nov. 1985, pp. 459-494.

22. V. Rajlich, "Comprehension and Evolution of Legacy Software," *Proc. of the 19th Int'l Conf. on Software Eng.*, Boston, Mass., May 1997, pp. 669-670.

23. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 2004.

24. T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proc. of the 5th Int'l Conf. on Software Eng.*, Baltimore, Md., May 93, pp. 482-498.

25. J. Singer, et al., "An Examination of Software Engineering Work Practices," *Proc. of the 1997 Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, Toronto, Ont., Nov. 1997, pp. 209-223.

26. K. Erdös and H.M. Sneed, “Partial Comprehension of Complex Programs (enough to perform maintenance),” *Proc. of the 6th Int’l Workshop on Program Comprehension*, Ischia, Italy, Jun. 1998, pp. 98-105.

27. Eclipse, “Eclipse.org home,” Jul. 2007; http://www.eclipse.org/.

28. Visual Studio, “Visual Studio 2005 Developer Center,” Jul. 2007; http://msdn2.microsoft.com/en-us/vstudio/default.aspx.

29. C. Ball, “What to Do When a Copy Is Not a Copy,” Oct. 25, 2006; http://www.law.com/jsp/legaltechnology/pubArticleLTN.jsp?id=1161680719761&rss=ltn.

30. CVS, “CVS - Open Source Version Control,” Jul. 2007; http://www.nongnu.org/cvs/.

31. Y. Liu and E. Stroulia, “Reverse Engineering the Process of Small Novice Software Teams,” *Proc. of the 10th Working Conf. on Reverse Eng.*, Victoria, B.C., Nov. 2003, pp. 102-103.

32. L. Zou and M.W. Godfrey, “Detecting Merging and Splitting Using Origin Analysis,” *Proc. of the 10th Working Conf. on Reverse Eng.*, Victoria, B.C., Nov. 2003, pp. 146-154.

33. OMG “Architecture-Driven Modernization Task Force,” Jul. 2007; http://adm.omg.org.

34. I. Pashov and M. Riebisch, “Using Feature Modeling for Program Comprehension and Software Architecture Recovery,” *Proc. of the 11th IEEE Int’l Conf. and Workshop on the Eng. of Computer-Based Systems*, May 2004, pp. 406-417.

35. Hibernate, “hibernate.org - Hibernate,” Jul. 2007; http://www.hibernate.org/.

36. Altova MapForce, “Data Mapping Tool from Altova: MapForce,” Jul. 2007; http://www.altova.com/products/mapforce/data_mapping.html.

37. S. Chamberlain, “An Enterprise Identifier Strategy for Global Naming Across Arbitrary C4I Systems,” *Proc. of the 6th Int’l Command and Control Research and Technology Symposium*, US Naval Academy, Annapolis, Md., Jun. 2001.

38. W. Arms, “Digital Libraries: Glossary (1999),” Jul. 2007; http://www.cs.cornell.edu/ wya/DigLib/MS1999/glossary.html.

39. J. Knodel and Martin Pinzger, “Improving Fact Extraction of Framework-Based Software Systems,” *Proc. of the 10th Working Conf. on Reverse Eng.*, Victoria, B.C., Nov. 2003, pp. 186-195.

40. R. Kazman, L. O'Brien, and C. Verhoef, "Architecture Reconstruction Guidelines, Third Edition," *Tech. Report CMU/SEI-2002-TR-034*, Carnegie Mellon Univ., Pittsburgh, Pa., Nov. 2003.

41. J. Jones, "Abstract Syntax Tree Implementation Idioms," *Proc. of the 10th Conf. on Pattern Languages of Programs*, Champaign, Ill., Sept. 2003.

42. C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," *Software - Practice & Experience*, vol. 25, no. 7, Jul. 1995, pp. 811-829.

43. Program-Transformation.Org, "Program Transformation Wiki / Decompilation Process," Jul. 2007; http://www.program-transformation.org/ Transform/DecompilationProcess.

44. J. Miecznikowski and L. Hendren, "Decompiling Java Using Staged Encapsulation," *Proc. of the 8th Working Conf. on Reverse Eng*., Stuttgart, Germany, Oct. 2001, pp. 368-374.

45. N.A. Naeem and L. Hendren, "Programmer-friendly Decompiled Java," *Proc. of the 14th Int'l Conf. on Program Comprehension*, Athens, Greece, Jun. 2006, pp. 327-336.

46. Q. Tu and M.W. Godfrey, "The Build-Time Software Architecture View," *Proc. of the 17th IEEE Int'l Conf. on Software Maintenance*, Florence, Italy, Nov. 2001, pp. 398-407.

47. GNU Make, "GNU Make - GNU Project - Free Software Foundation (FSF)," Jul. 2007; http://www.gnu.org/software/make/.

48. Apache Ant, "Apache Ant - Welcome," Jul. 2007; http://ant.apache.org/.

49. H.A. Müller, et. al., "Reverse Engineering: A Roadmap," *Proc. of the 22nd Int'l Conf. on Software Eng. (ICSE'2000)*, Limerick, Ireland, June 2000, pp. 47-60.

50. G.A. Di Lucca, A.R. Fasolino, and U. de Carlini, "Recovering Class Diagrams from Data-Intensive Legacy Systems," *Proc. of the Int'l Conf. on Software Maintenance (ICSM'00)*, San Jose, Calif., Oct. 2000, pp. 52-63.

51. J.-L. Hainaut, et al., "Structure Elicitation in Database Reverse Engineering," *Proc. of the 3rd Working Conf. on Reverse Eng.*, Monterey, Calif., Nov. 1996, pp. 131-140.

52. A. Seesing and A. Orso, "InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse," *Proc. of the 2005 OOPSLA Workshop on Eclipse Technology eXchange 2005*, San Diego, Cal., Oct. 2005, pp. 45-49.

53. J. Guitart, et al., “Performance Analysis Tools for Parallel Java Applications on Shared-Memory Systems,” *Proc. of the Int’l Conf. on Parallel Processing*, Valencia, Spain, Sept. 2001, pp. 357-364.

54. M. Biberstein, et al., “Instrumenting Annotated Programs,” *Proc. of the 1st ACM/USENIX Int’l Conf. on Virtual Execution Environments*, Chicago, Ill., Jun. 2005, pp. 164-174.

55. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Press, 1990.

56. J. Guo, Y. Liao, and B. Parviz, “A Performance Validation Tool for J2EE Applications,” *Proc. of the 13th Ann. IEEE Int’l Symp. and Workshop on Eng. of Computer Based Systems*, Potsdam, Germany, Mar. 2006, pp. 387-396.

57. BCEL, “BCEL - Byte Code Engineering Library (BCEL),” Jul. 2007; http://jakarta.apache.org/bcel/manual.html.

58. Eclipse TPTP, “Eclipse Test & Performance Tools Platform Project,” Jul. 2007; http://www.eclipse.org/tptp/.

59. JUnit, “JUnit, Testing Resources for Extreme Programming,” Jul. 2007; http://www.junit.org/.

60. Eclipse BIRT, “Eclipse BIRT Home,” Jul. 2007; http://www.eclipse.org/birt/phoenix/.

61. FileMon, “FileMon for Windows v7.04,” Jul. 2007; http://www.microsoft.com/technet/sysinternals/utilities/Filemon.mspx.

62. B.S. Bloom, *Taxonomy of Educational Objectives, Handbook 1: The Cognitive Domain*, David McKay Co, 1956.

63. N.E. Fenton, “Software Measurement Programs,” *Software Testing and Quality Eng.*, vol. 1, no. 3, 1999, pp. 40-46.

64. Headway Software, *Headway reView/DesignKeeper 3.4: User Documentation*, Headway Software, 2003.

65. S.R. Chidamber and C.F. Kemerer, “Towards a Metrics Suite for Object Oriented Design,” *IEEE Trans. on Software Eng*. vol. 20, no. 6, June 1994, pp. 476-493.

66. R. Martin, “OO Design Quality Metrics: An Analysis of Dependencies,” *Proc. of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, Oct. 1994.

67. A.D. Eisenberg and K. De Volder, “Dynamic Feature Traces: Finding Features in Unfamiliar Code,” *Proc. of the 21st IEEE Int’l Conf. on Software Maintenance*, Budapest, Hungary, Sept. 2005, pp. 337-346.

68. N. Wilde and M.C. Scully, “Software Reconnaissance: Mapping Features to Code,” *Software Maintenance: Research and Practice*, vol. 7, no. 1, 1995, pp. 49-62.

69. T. Eisenbarth, R. Koschke, and D. Simon, “Aiding Program Comprehension by Static and Dynamic Feature Analysis,” *Proc. of the IEEE Int’l Conf. on Software Maintenance (ICSM’01)*, Florence, Italy, Nov. 2001, pp. 602-611.

70. G. Birkoff, *Lattice Theory*, American Mathematical Society, 1992.

71. R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society, 1991.

72. K. Beck and R. Johnson, “Patterns Generate Architectures,” *Proc. of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, Jul. 1994, pp. 139-149.

73. J. Hutchinson and G. Kotonya, “Patterns and Component-Oriented System Development,” *Proc. of the 31st EUROMICRO Conf. on Software Eng. and Advanced Applications*, Porto, Portugal, Aug. 2005, pp. 126-133.

74. E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

75. W. Wang and V. Tzerpos, “DPVK - An Eclipse Plug-in to Detect Design Patterns in Eiffel Systems,” *Electronic Notes in Theoretical Computer Science*, vol. 107, Dec. 2004, pp. 71-86.

76. Architecture Board ORMSC, “Model Driven Architecture (MDA),” *Document No. ORMSC/2001-07-01*, Object Management Group, Jul. 2001.

77. J. Miller and J. Mukerji, “MDA Guide Version 1.0.1,” *Document No. omg/2003-06-01*, Object Management Group, Jun. 2003.

78. R. Soley, “Model Driven Architecture,” Object Management Group, Nov. 2000.

79. V. Tzerpos, *Comprehension-Driven Software Clustering*, Ph.D. Thesis, University of Toronto, Toronto, Ont., 2001.

80. B. Andreopoulos, et al., “Multiple Layer Clustering of Large Software Systems,” *Proc. of the 12th Working Conf. on Reverse Eng*, Pittsburgh, Pa., Nov. 2005, pp. 79-88.

81. V. Tzerpos and R.C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering," *Proc. of the 7th Working Conf. on Reverse Eng.*, Brisbane, Australia, Nov. 2000, pp. 258-267.

82. S. Mancoridis, et al., "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *Proc. of the IEEE Int'l Conf. on Software Maintenance (ICSM'99)*, Oxford, England, Aug. 1999, pp. 50-59.

83. B.S. Mitchell and S. Mancoridis, "Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements," *Proc. of Int'l Conf. of Software Maintenance (ICSM'01)*, Florence, Italy, Nov. 2001, pp. 744-753.

84. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Professional, 2005.

85. Visual Paradigm, "UML 2 Diagrams - UML Modeling Tool," Jul. 2007; http://www.visual-paradigm.com/VPGallery/diagrams/.

86. G.C. Murphy, D. Notkin, and E.S.-C. Lan, "An Empirical Study of Static Call Graph Extractors," *Proc. of the 18th Int'l Conf. on Software Eng.*, Berlin, Germany, Mar. 1996, pp. 90-99.

87. Rational PurifyPlus, "IBM - Rational PurifyPlus - Rational PurifyPlus - Software," Jul. 2007; http://www-306.ibm.com/software/awdtools/purifyplus/win/.

88. B.B. Bederson, et al., "PAD++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics," *J. of Visual Languages and Computing*, vol. 7, no. 1, 1996, pp. 3-32.

89. K.L. Summers, et al., "An Experimental Evaluation of Continuous Semantic Zooming in Program Visualization," *2003 IEEE Symposium on Information Visualization*, Seattle, Wash., Oct. 2003.

90. E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, Jan. 1990, pp. 13-17.

91. A. van Deursen and T. Kuipers, "Building Documentation Generators," *Proc. of the 15th IEEE Int'l Conf. on Software Maintenance (ICSM'99)*, Leicester, UK, Aug. 1999, pp. 40-49.

92. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE Press, 2000.

93. P. Brown, "Integrated Hypertext and Program Understanding Tools," *IBM Systems J.* vol. 30, no. 3, 1991, pp. 363-392.

94. C. de Oliveira Braga et. al., “Documentu: A Flexible Architecture for Documentation Production Based on a Reverse-engineering Strategy,” *J. of Software Maintenance: Research and Practice*, vol. 10, no. 4, Jul./Aug. 1998, pp. 279-303.

95. V. Rajlich, “Incremental Redocumentation Using the Web,” *IEEE Software*, vol. 17, no. 5, Sept./Oct. 2000, pp. 102-106.

96. P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley Professional, 2000.

97. G.A. Di Lucca, A.R. Fasolino, and U. de Carlini, “Recovering Use Case Models from Object-Oriented Code: a Thread-based Approach,” *Proc. of the 7th Working Conf. on Reverse Eng.*, Brisbane, Australia, Nov. 2000, pp. 108-117.

98. Department of Defense, *DoD Architecture Framework Version 1.5: Volume I: Definitions and Guidelines*, 2007.

99. W.G. Wood, et al., “DoD Architecture Framework and Software Architecture Workshop Report,” *Tech. Note CMU/SEI-2003-TN-006*, Carnegie Mellon Univ., Pittsburgh, Pa., Mar. 2003.

100. *IEEE Standard for Information Technology - Software Life Cycle Processes*, IEEE/EIA 12207.0-1996, IEEE Press, 1996.

101. Moore, J. “ISO 12207 and Related Software Life-Cycle Standards,” Jul. 2007; http://www.acm.org/tsc/lifecycle.html.

102. International Standards Organization, “Information technology - Guide for ISO/IEC 12207 (Software Life Cycle Processes),” *Tech. Report ISO/IEC TR 15271:1998*, 1998.

103. COPlanS, “COPlanS - Collaborative Operations Planning System,” Jul. 2007; http://www.valcartier.drdc-rddc.gc.ca/poolpdf/e/166_e.pdf.

104. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.

105. J.C. Roberts, “On Encouraging Multiple Views for Visualization,” *Proc. of the Int’l Conf. on Information Visualization (IV’98)*, London, UK, Jul. 1998, pp. 8-14.

106. Eclipse CDT, “Eclipse C/C++ Development Tooling - CDT,” Jul. 2007; http://www.eclipse.org/cdt/.

10. Appendix A

Concept analysis is a mathematical technique that provides insights into binary relations [70]. It is based on a relation *R* between a set of objects *O* and a set of attributes *A*, i.e., $R \subseteq O \times A$. As an example, Table 2 shows the binary relation between arbitrary objects and attributes. In this case, an object o_i has attribute a_j if row *I* and column *j* are marked with an *x*.

Table 2. *Relation Example [69]*

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
o_1	x	x						
o_2			x	x	x			
o_3			x	x		x	x	x
o_4			x	x	x	x	x	x

A pair *(O, A)* is called a concept if all the objects in *O* share all attributes in *A*. Table 3 contains the concepts for the relation in Table 2.

Table 3. *Concepts for Table 2*

c_1	$(\{o_1, o_2, o_3, o_4\}, \emptyset)$
c_2	$(\{o_2, o_3, o_4\}, \{a_3, a_4\})$
c_3	$(\{o_1\}, \{a_1, a_2\})$
c_4	$(\{o_2, o_4\}, \{a_3, a_4, a_5\})$
c_5	$(\{o_3, o_4\}, \{a_3, a_4, a_6, a_7, a_8\})$
c_6	$(\{o_4\}, \{a_3, a_4, a_5, a_6, a_7, a_8\})$
c_7	$(\emptyset, \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\})$

If $c_1 \leq c_2$ holds, then c_1 is said to be a subconcept of c_2 and c_2, a superconcept of c_1. For example, since $(\{o_2, o_4\}, \{a_3, a_4, a_5\}) \leq (\{o_2, o_3, o_4\}, \{a_3, a_4\})$, then c_4 is a subconcept of c_2 and c_2 is a superconcept of c_4.

The set of all concepts in Table 3 is called a concept lattice. It is graphically represented as a directed acyclic graph, where a node represents a concept and an edge denotes a superconcept/subconcept relation. Figure 36 shows the concept lattice for the relation in Table 2. The most general concept, denoted by T, is called the top element. The most special concept, denoted by $\perp$, is called the bottom element.

The concept lattice of Figure 36 can be visualized in a more readable manner by marking a graph node with an attribute $a \in A$, i.e., $\mu(a)$, if it represents the most general concept that has a in its intent. Analogously, a node will be marked with an object $o \in O$, i.e., $\gamma(o)$, if it represents the most special concept which has o in its extent. The resulting representation is called a sparse representation of the lattice. Figure 37 displays the equivalent sparse representation of Figure 36.

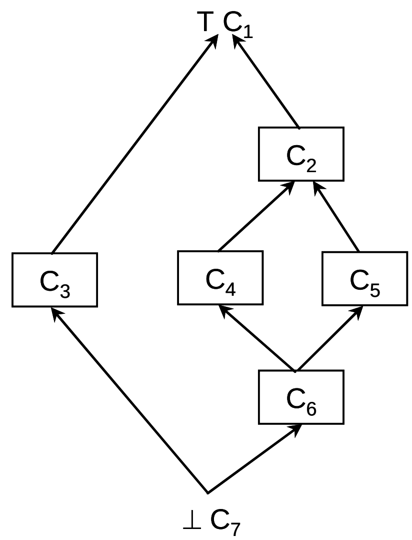


Figure 36. *Concept Lattice*

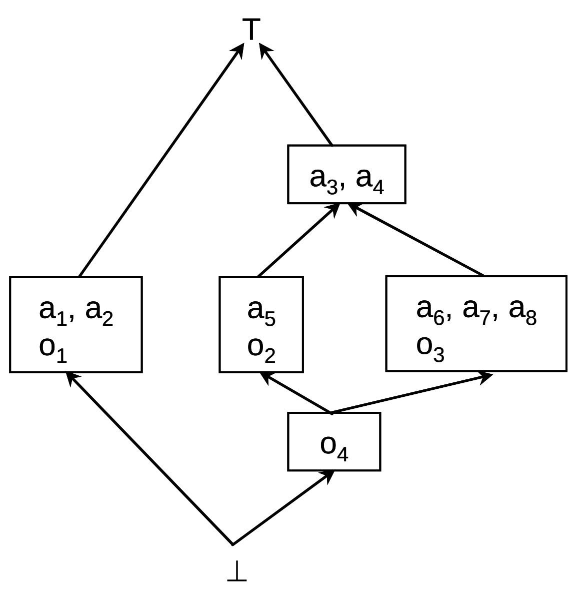


Figure 37. *Sparse Representation*

In a sparse representation, the content of a node N can be derived as follows [69]:

- The objects of N are all objects at and below N.
- The attributes of N are all attributes at and above N.

To derive scenario-method relationships using concept analysis, one has to define a formal context. In the present case, methods will be considered objects and scenarios, attributes. As a result, a pair (*method s*, *scenario S*) is in relation R if s is executed when S is performed [69].

The process of locating features using concept analysis is as follows [69]:

1. The set of relevant features is identified $F = \{f_1 \ldots f_n\}$
2. The scenarios $A = \{S_1 \ldots S_q\}$ are identified so that the features in F are covered.
3. Execution traces are generated so that all required methods $O = \{S_1 \ldots S_q\}$ for each scenario are identified.
4. The relation table R is created so that $(S_1, s_1), (S_1, s_2), \ldots, (S_q, s_p) \in R$
5. Concept analysis is performed for (O, A, R)

Concept analysis applied to the formal context yields a lattice from which the following relationships can be derived [69]:

- A method s is required for all scenarios at and above $\gamma(s)$.
- A scenario S requires all methods at and below $\mu(S)$.
- A method s is specific to exactly one scenario S if S is the only scenario on all paths from $\gamma(s)$ to the top element.
- A scenario S is specific to exactly one method s if s is the only method on all paths from $\mu(S)$ to the bottom element.
- Scenarios to which two methods s_1 and s_2 jointly contribute can be identified by $\gamma(s_1) \vee \gamma(s_2)$. In the lattice, it is the closest common node toward the top element starting at the nodes to which s_1 and s_2 are attached. All scenarios at and above this common node are those jointly implemented by s_1 and s_2.
- Methods jointly required for two scenarios S_1 and S_2 are described by $\mu(S_1) \wedge \mu(S_2)$. In the lattice, it is the closest common node towards the bottom element starting at the nodes to which S_1 and S_2 are attached. All

methods at and below this common node are those jointly required for S_1 and S_2.

- Methods required for all scenarios can be found at the bottom element.

- Scenarios that require all methods can be found at the top element.

- If $\gamma(s_1) < \gamma(s_2)$ holds for two methods s_1 and s_2, then method s_2 is more specific with respect to the given scenario than method s_1 because s_1 contributes not just to the features for which s_2 contributes, but also to other features.

- If $\mu(S_1) < \mu(S_2)$ holds for two scenarios S_1 and S_2, then scenario S_2 is based on scenario S_1 because if S_2 is executed, all methods in the extent of $\mu(S_1)$ need also to be executed.

Based on the relationships derived from the concept lattice, a decision can be taken to analyze only a subset of the original features in depth due to the additional dependencies that concept analysis could reveal. All the methods required for these features form a starting point for further static analyses [69].

List of Acronyms

A	Abstractness
ACDC	Algorithm for Comprehension-Driven Clustering
ADM	Architecture-Driven Modernization
API	Application Programming Interface
AST	Abstract Syntax Tree
AV	All View
BCEL	Byte Code Engineering Library
BIRT	Business Intelligence and Reporting Tools
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance
Ca	Afferent Coupling
CBO	Coupling Between Object Classes
CC	Cyclomatic Complexity
CDT	C/C++ Development Tooling
Ce	Efferent Coupling
CF	Canadian Forces
COPlanS	Collaborative Operations Planning System
CORBA	Common Object Request Broker Architecture
CVS	Concurrent Versions System
DIT	Depth of Inheritance Tree
DMS	Distance from the Main Sequence
DND	Department of National Defence
DoDAF	Department of Defense Architecture Framework

DRDC	Defence Research and Development Canada
EDI	Environnement de Développement Intégré
EID	Enterprise Identifier
ER	Entity Relationship
FC	Forces canadiennes
GIOP	General Inter-ORB Protocol
GXL	Graph eXchange Language
I	Instability
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
JDT	Java Development Tools
LCOM	Lack of Cohesion
LOC	Lines of Code
MDA	Model Driven Architecture
NCNB	Non-Comment Non-Blank
NOC	Number of Children
OASIS	Opening up Architecture of Software-Intensive Systems
OASIS	Ouverture d'Architectures de Systèmes Informatisés Significativement
OMG	Object Management Group
ORB	Object Request Broker
OV	Operational View
PIM	Platform Independent Model
PSM	Platform Specific Model
RFC	Response for a Class

RUP	Rational Unified Process
SdS	Systèmes de systèmes
SLOC	Source Lines of Code
SoS	System of Systems
SV	Systems View
TPTP	Test and Performance Tools Platform
TV	Technical Standards View
UML	Unified Modeling Language
WMC	Weighted Methods per Class
XML	Extensible Markup Language

Distribution List

INTERNAL DISTRIBUTION

1 - Director General

3 - Document Library

1 - Head, System of Systems

1 - Philippe Charland (author)

1 - David Ouellet (author)

1 - Dany Dessureault (author)

1 - Michel Lizotte (author)

1 - Geneviève Dussault

1 - Michel Ducharme

1 - Head, Information and Knowledge Management

1 - François Lemieux

1 - Martin Salois

1 - Head, Decision Support Systems

1 - Bruno Gilbert

1 - Marc Lauzon

1 - Marc Grondin

1 - LCol Pierre Lefebvre

EXTERNAL DISTRIBUTION

1 - DRDKIM (PDF file)

DRDC Headquarters - 305 Rideau Street, Ottawa, ON, K1A 0K2

1 - Director Science and Technology Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (DSTC4ISR)

1 - Klaus Kollenberg (DSTC4ISR 3)

1 - Donna Wood (DSTC4ISR 4)

1 - Norbert Haché (DSTC4ISR SPO)

1 - Richard Lestage (Director Science and Technology Air 6)

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

(Highest Classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

1. ORIGINATOR (name and address) Defence Research and Development Canada Valcartier 2459, Pie-XI Blvd North Québec, Québec G3J 1X5 Canada	2. SECURITY CLASSIFICATION (Including special warning terms if applicable) Unclassified

3. TITLE (Its classification should be indicated by the appropriate abbreviation (S, C, R or U)

Opening up architectures of software-intensive systems: A functional decomposition to support system comprehension (U)

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Charland, Philippe; Ouellet, David; Dessureault, Dany; Lizotte, Michel

5. DATE OF PUBLICATION (month and year) October 2007	6a. NO. OF PAGES 90	6b .NO. OF REFERENCES 106

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (name and address)

Defence Research and Development Canada Valcartier

2459, Pie-XI Blvd North

Val-Bélair, Québec

G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (Please specify whether project or grant) 15ak	9b. CONTRACT NO.
10a. ORIGINATOR'S DOCUMENT NUMBER DRDC Valcartier TM 2006-732	10b. OTHER DOCUMENT NOS N/A

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

- [x] Unlimited distribution
- [ ] Restricted to contractors in approved countries (specify)
- [ ] Restricted to Canadian contractors (with need-to-know)
- [ ] Restricted to Government (with need-to-know)
- [ ] Restricted to Defense departments
- [ ] Others

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

With the increasing needs of the Canadian Forces (CF) for systems interoperability, techniques and tools have to be developed in order to build systems of systems (SoS), i.e., systems whose components are themselves independent systems from an operational and managerial viewpoint. However, before existing systems can interoperate, their architectures first need to be recovered and comprehended. This technical memorandum describes the functional decomposition of an integrated suite of tools to assist with software system architecture recovery and comprehension. It was designed based on the requirements already identified in the scientific literature for comprehension tools, on a qualitative study conducted using existing tools, as well as on a state-of-the-art survey on system architecture recovery and comprehension. Following the conception of this functional decomposition, a prototype implementing it will be developed into an integrated development environment (IDE) to assist the CF in recovering and comprehending the architecture of already existing software systems.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software architecture recovery, program comprehension, program understanding tools, reverse engineering, functional decomposition.

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

DEFENCE R&D DÉFENSE

WWW.drdc-rddc.gc.ca